# Structures and Processes for Managing Model-Metamodel Co-evolution

Louis Mathew Rose

Department of Computer Science
University of York

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy.

July 2011

# Abstract

Software changes over time. During the lifetime of a software system, unintended behaviour must be corrected and new requirements satisfied. Because software changes are costly, tools for automatically managing change are commonplace. Contemporary software development environments can automatically perform change management tasks such as impact analysis, refactoring and background compilation.

Increasingly, models and modelling languages are first-class citizens in software development. Model-Driven Engineering (MDE), a state-of-the-art approach to software engineering, prescribes the use of models throughout the software engineering process. In MDE, modelling tools and task-specific language are used to generate an ultimate artefact, such as simulation models or working code.

Contemporary MDE environments provide little support for managing a type of evolution termed *model-metamodel co-evolution*, in which changes to a modelling language are propagated to models. This thesis demonstrates that model-metamodel co-evolution occurs often in MDE projects, and that dedicated structures and processes for its management can increase developer productivity. Structures and processes for managing model-metamodel co-evolution are proposed, developed, and then evaluated by comparison to existing structures and processes with quantitative and qualitative techniques.

For my Nanna Spence

# Contents

# List of Figures

# List of Tables

# Listings

# Acknowledgements

I thank my supervisors, Prof. Richard Paige and Dr. Fiona Polack, for their invaluable guidance, support and encouragement throughout my doctoral research. I am also very grateful to Dr. Dimitrios Kolovos for his advice and for countless rewarding discussions. I thank Dr. Simon Poulding for reviewing a technical report of preliminary research results.

I would like to thank my colleagues in the department for their support and friendship, particularly Dr. Heather Barber, Mark Read and Tara Gilliam. For many interesting and entertaining discussions, I thank my colleagues and friends in the Enterprise Systems group. I am also grateful to Dr. Anne Etien, Dr. Antonio Cicchetti, Markus Herrmannsdörfer and Dr. Kelly Garcés for the interesting debates and discussions that we have shared.

I would like to express gratitude to my parents and my brother Nathan for their continued support and reassurance. Finally, I am eternally grateful to my partner, Kate Senior, for her patience and optimism, and for our many adventures together.

# Author Declaration

Except where stated, all of the work contained in this thesis represents the original contribution of the author. Section 6.3 reports collaborative experiments with model migration tools, and that section makes clear the roles of the author and other participants.

Parts of the work described in this thesis have been previously published by the author in:

- **The Epsilon Generation Language**, Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, volume 5095 of LNCS, pages 1-16. Springer, 2008,
[Rose *et al.* 2008b].
- **Constructing Models with the Human-Usable Textual Notation**, Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of LNCS, pages 249-263. Springer, 2008, [Rose *et al.* 2008a].
- **An Analysis of Approaches to Model Migration**, Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A.C. Polack in *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop*, co-located with MoDELS 2009, [Rose *et al.* 2009b].
- **Enhanced Automation for Managing Model and Metamodel Inconsistency**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack in *Proc. International Conference on Automated Software Engineering (ASE)*, pages 545-549, ACM Press, 2009, [Rose *et al.* 2009a].
- **Concordance: An Efficient Framework for Managing Model Integrity**, Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James. R. Williams, Richard F. Paige, Fiona A.C. Polack, and Kiran J. Fernandes in *Proc. European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6138 of LNCS, pages 62-73. Springer, 2010, [Rose *et al.* 2010c].

- **Model Migration with Epsilon Flock**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 6142 of LNCS, pages 184-198. Springer, 2010, [Rose *et al.* 2010f].
- **Model Migration Case**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. Transformation Tools Contest (TTC)*, co-located with TOOLS Europe 2010, [Rose *et al.* 2010e].
- **Migrating Activity Diagrams with Epsilon Flock**, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack in *Proc. Transformation Tools Contest (TTC)*, co-located with TOOLS Europe 2010, [Rose *et al.* 2010d].
- **A Comparison of Model Migration Tools**, Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A.C. Polack in *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6394 of LNCS, pages 6175. Springer, 2010, [Rose *et al.* 2010b].

# Chapter 1

# Introduction

Today's software engineers build distributed and interoperating systems with sophisticated graphical interfaces rather than the insular, monolithic, and command-line driven mainframe applications built by their predecessors. For example, the NatWest and Royal Bank of Scotland banking systems were successfully unified in 2003, which involved processing 14 million customer records, 13 million account records and 22 million direct debits in a single weekend [RAE & BCS 2004, pg26]. Distributed and interoperable systems are key requirements in the National Programme for IT[1], which seeks to modernise the United Kingdom's National Health Service with computerised systems for managing the nation's patient records. In the United States of America, the goals of the Department of Defense depend on increasingly complex systems, which encompass "thousands of platforms, sensors, decision nodes, weapons, and war-fighters connected through heterogeneous wired and wireless networks" [Northrop 2006].

Some of the software demanded by users and developers today is so complicated that its construction is not possible, even using state-of-the-art software engineering techniques [Selic 2003]. For example, a large supermarket chain recently planned to develop software for managing a new loyalty card scheme. However, implementing the system would have involved devising algorithms for efficiently searching 4 terabytes of customer data, and it was deemed impossible to implement despite its obvious commercial advantages [RAE & BCS 2004, pg15]. Demand, however, does not appear to exceed capability in all areas of computer science.

Hardware development, for example, seems to advance more quickly than software development. Each year, faster personal computers with larger disk drives become available, while operating systems, office software and development environments seem to improve more gradually. Radical advances in software development appear to occur only by raising the level of abstraction at which software is specified [Brooks Jr. 1987, Selic 2003, Kleppe *et al.* 2003]. Improvements to the training and education of software engineers have also been suggested as key for the construction of increasingly complex software systems [RAE & BCS 2004].

---

[1] `http://www.connectingforhealth.nhs.uk/about/benefits/statement0607.pdf`

## 1.1 An Overview of Model-Driven Engineering

Historically, raising the level of abstraction of software development has led to increased productivity [Brooks Jr. 1987, Barry 2006, Kelly & Tolvanen 2008]. For example, assembly language provides mnemonics for machine code, allowing developers to disregard extraneous detail (such as the binary representation of instructions). Object-orientation and functional programming permit further abstraction over assembly language, enabling developers to express solutions in a manner that is more representative of their problem domain.

Model-Driven Engineering (MDE) is a contemporary approach to software engineering that seeks to abstract away from technological details (such as programming languages and off-the-shelf software components) and towards the problem domain of the system (for example: accounting, managing patient records, or searching the Internet) [Frankel 2002, Kleppe *et al.* 2003, Selic 2003]. To this end, MDE prescribes, throughout the software engineering process, the use of models to capture the relevant details of the problem domain. Software development is driven by manipulating (transforming, validating, merging, comparing, etc.) the models to automatically generate an ultimate artefact, such as working code or simulation models.

MDE reportedly provides many benefits over traditional approaches to software engineering. Conclusions drawn from two unpublished case studies suggest that MDE can lead to increased productivity by reducing the amount of time to develop a system, and by reducing the number of defects discovered throughout development [Watson 2008]. MDE can be used to increase the productivity of software development and the maintainability of software systems [Kleppe *et al.* 2003]. Separating platform-specific and platform-independent details using MDE can facilitate greater portability of systems [Frankel 2002]. Section 2.5.1 discusses further benefits of MDE.

Notwithstanding its benefits, MDE introduces additional challenges for software development. Large models are commonplace in many software engineering projects, and contemporary MDE environments have not be optimised to facilitate the manipulation of large models [Kolovos *et al.* 2008a]. More generally, MDE introduces new challenges for managing change throughout the lifetime of a system [Mens & Demeyer 2007]. This thesis focuses on the latter challenge, which is part of a branch of computer science termed *software evolution*.

## 1.2 An Overview of Software Evolution

Software changes over time. During the lifetime of a software system, unintended behaviour must be corrected and new requirements satisfied. Because modern software systems are rarely isolated from other systems, changes are also made to facilitate interoperability with new systems [Sjøberg 1993].

*Software evolution* is an area of computer science that focuses on the way in which a software system changes over time in response to external pressures (such as changing requirements, or the discovery of unintended behaviour). The terms software evolution and software maintenance are used interchangeably in the software engineering literature. In this thesis, *evolution* is preferred to *maintenance*, because the latter can imply deterioration caused by repeated use, and most software systems do not deteriorate in this manner [Ramil & Lehman 2000]. Other than sharing some terminology, software evolution is not related to evolutionary algorithms, a branch of computer science that encompasses genetic programming and genetic algorithms.

In the past, studies have suggested that software evolution can account for as much as 90% of a development budget [Erlikh 2000, Moad 1990], and there is no reason to believe that the situation is different today. Although such figures have been described as uncertain [Sommerville 2006, ch. 21], precise figures are not required to demonstrate that the effects of evolution can inhibit the productivity of software development.

For example, suppose that we are developing a software system using a combination of hand-written code and off-the-shelf components. Part way through development, one of the components changes to support a new requirement. When using the new version of the component, we must first determine whether our system exhibits any unintended behaviour, identify the cause of the unintended behaviour, and change the system accordingly. The resources allocated to correcting any unintended behaviour are not being used to develop features for the users of our system.

Primarily, software evolution research seeks to reduce the cost of making changes to a system. Analysis of the effects of evolution facilitates decision making. For example, analysis of our system might indicate that using a new version of a component will introduce three defects, but simplify the implementation of two features. Studying the way in which systems evolve leads to improvements in software development tools and processes that reduce the effects of evolution. For example, contemporary software development environments recognise compilation as a common activity during software evolution, and often perform automatic and incremental compilation of source code in the background. Future changes to a system might be anticipated by identifying the ways in which the system has previously evolved. For example, understanding the ways in which a system has been affected by using a new version of a component might highlight ways in which the system can be better protected against changes to its dependencies.

## 1.3 Motivation: Software Evolution in MDE

Proponents of MDE suggest that, compared to traditional approaches to software engineering, application of MDE leads to systems that better support evolutionary change [Kleppe *et al.* 2003]. Large-scale systems developed with traditional approaches to software engineering have been described as examples of a modern-day Sisyphus[2], whose developers must constantly perform evolution to support conformance to changing standards and interoperability with external systems [Frankel 2002]. Some proponents suggest that MDE can be used to reduce the cost of software evolution [Frankel 2002],

---

[2] In Greek mythology, Sisyphus was condemned to an eternity of repeatedly rolling a boulder to the top of a mountain, only to see it return to the mountain's base.

while others report that MDE introduces additional challenges for managing software evolution [Mens & Demeyer 2007].

In particular, the evolution of models, modelling languages and other MDE development artefacts must be managed in MDE. Contemporary development environments provide some assistance for performing software evolution activities (by, for example, providing transformations that automatically restructure code). However, there is little support for software evolution activities that involve models and modelling languages. Chapters 3 and 4 review, analyse and motivate improvements to the way in which software evolution is identified and managed in contemporary MDE development environments. Chapters 5 and 6 explore the extent to which the productivity of identifying and managing evolutionary change can be increased by extending contemporary MDE development environments with additional, dedicated structures and processes.

## 1.4 Research Hypothesis and Method

The research presented in this thesis explores the hypothesis below. The emboldened terms are potentially ambiguous, and their definition follows the hypothesis.

> *In existing MDE projects, the evolution of **MDE development artefacts** is typically managed in an ad-hoc manner with little regard for re-use. Dedicated structures and processes for **managing evolutionary change** can be designed by analysing evolution in existing MDE projects. Furthermore, supporting those dedicated structures and processes in contemporary MDE environments is beneficial in terms of increased **productivity** for software development activities pertaining to the management of evolutionary change.*

In this thesis, the terms below have the following definitions:

**MDE development artefacts.** Compared to traditional approaches to software engineering, MDE uses additional development artefacts as first-class citizens in the development process. The additional development artefacts peculiar to MDE include models and modelling languages, as well as model management operations (such as model transformations). Chapter 2 describes models, modelling languages and model management operations in more detail.

**Managing evolutionary change.** Contemporary computer systems are constructed by combining numerous interdependent artefacts. Evolutionary changes to one artefact can affect other artefacts. For example, changing a database schema might cause data to become invalid with respect to the database integrity constraints, and changing source code may require recompilation of object code to ensure the latter is an accurate representation of the former. Managing evolutionary change typically comprises three related activities: *identifying* when a change has occurred, *reporting* the effects of a change, and *reconciling* affected artefacts in response to a change. Chapter 3 reviews existing approaches to managing evolutionary change.

**Productivity** is a measure of the output from a process, per unit of input to that process [Beattie *et al.* 2007]. For example, the productivity of data entry might be measured by counting the number of characters produced per typist per hour. An Optical Character Recognition (OCR) system might increase data entry productivity, but this is likely to be dependent on many factors, including: the accuracy and capabilities of the OCR system, the speed and accuracy of each typist, and the legibility and consistency of the data. Managing and measuring the productivity of software engineering is challenging. Division of labour, for example, can decrease productivity in software engineering as evidenced by Brooks's eponymous law ("adding manpower to a late software project makes it later") [Brooks Jr. 1995]. This thesis investigates the productivity of small, well-defined software development activities, and not the productivity of software engineering projects.

### 1.4.1 Thesis Objectives

The objectives of the thesis are to:

1. Identify and analyse the evolution of MDE development artefacts in existing projects.
2. Investigate the extent to which existing structures and processes can be used to manage the evolution of MDE development artefacts.
3. Propose and develop prototypes of new structures and processes for managing the evolution of MDE development artefacts, and integrate those structures and processes with a contemporary MDE development environment.
4. Evaluate the proposed structures and processes for managing evolutionary change, particularly with respect to productivity.

### 1.4.2 Research Method

To explore the hypothesis outlined above, the thesis research was conducted using the method described in this section and summarised in Figure 1.1. The shaded boxes represent the three *phases* of research, which are described below. The unshaded boxes represent inputs and outputs to those phases.

Firstly, the *analysis* phase involved studying the evolution of MDE development artefacts in existing projects. The results of the analysis phase were used to determine a category of evolution that lacked support in contemporary MDE development environments, *model-metamodel co-evolution* or, simply *co-evolution*. Co-evolution examples from existing MDE projects were used to categorise existing processes for managing co-evolution, and to formulate requirements for new structures and processes

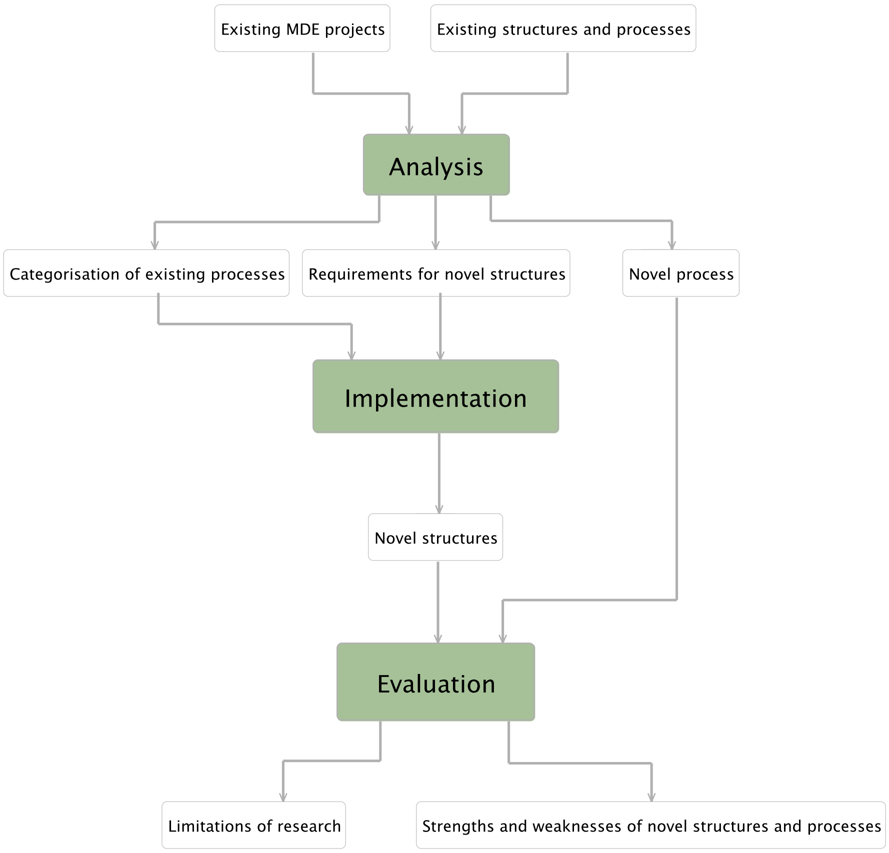


Figure 1.1: Overview of the research method.

for managing co-evolution. The analysis phase also led to the identification of *user-driven co-evolution*, a process for managing co-evolution that has not previously been recognised in the co-evolution literature.

The *implementation* phase involved proposing, designing and implementing prototypes of novel structures for managing co-evolution, and integrating the prototypical structures with a contemporary MDE environment. The co-evolution examples identified in the analysis phase were used for testing the implementation of the structures.

The *evaluation* phase involved assessing the novel structures for managing co-evolution by comparison to existing structures, and demonstrating the novel process. Evaluation was performed using further examples of co-evolution. To mitigate a possible threat to the validity of the research, the examples used in the evaluation phase were different from those used in the analysis phase. The strengths and weaknesses of the novel and existing structures and processes were synthesised from the comparisons, particularly with respect to the productivity of the development activities that are used to manage co-evolution.

A similar method was successfully used to explore the extent to which component-based applications can be automatically evolved [Dig 2007]. Initially, *analysis* was conducted to identify and categorise evolution in five existing component-based applications, with the hypothesis that many of the changes could be classified as behaviour-preserving [Dig & Johnson 2006b]. Examples from the survey facilitated *implementation* of a novel algorithm for the automatic detection of behaviour-preserving changes [Dig *et al.* 2006]. The algorithm was used to implement tools for migrating code in a distributed software development environment [Dig & Johnson 2006a], and for analysing the history of component-based applications [Dig *et al.* 2007]. The latter facilitated better understanding of program evolution, and refinement of the detection algorithm. Finally, *evaluation* of the tools and detection algorithm was performed by application to three further component-based applications [Dig 2007].

## 1.5 Results of the Thesis Research

This thesis proposes novel structures and processes for managing model-metamodel co-evolution. Prototypical reference implementations of the proposed structures have been constructed, including *Epsilon HUTN* (a textual modelling notation) and *Epsilon Flock* (a model migration language). The reference implementations have extended and reused Epsilon [Kolovos 2009], an extensible platform for specifying MDE languages and tools, and are interoperable with the Eclipse Modelling Framework [Steinberg *et al.* 2008], arguably the most widely used MDE modelling framework.

Additionally, this thesis identifies a novel process for managing model-metamodel co-evolution and proposes a theoretical categorisation of existing process for managing model-metamodel co-evolution. The novel process, termed *user-driven co-evolution*, is demonstrated by application to a MDE development process for a real-world project.

The research hypothesis has been validated by comparing the prototypes of the proposed structures and processes with existing structures and processes using examples of evolution from real-world MDE projects. Evaluation has been performed using several approaches, including a collaborative comparison of model migration tools carried out with three MDE experts, comparing quantitive measurements of the proposed and existing migration languages, and application of the proposed structures and processes to two examples of evolution, including an example from a widely used modelling language, the Unified Modelling Language (UML) [OMG 2007a]. The evaluation also explored areas in which the prototypical implementations of the proposed structures and processes might be usefully improved to be fit for industrial use.

## 1.6 Thesis Structure

Chapter 2 gives an overview of MDE by defining terminology; describing associated engineering principles, practices and tools; and reviewing related areas of computer science. Section 2.5 synthesises some of the benefits of, and challenges for, contemporary MDE.

Chapter 3 reviews theoretical and practical software evolution research. Areas of research that underpin software evolution are described, including refactoring, design patterns, and traceability. The review then discusses work that approaches particular categories of evolution problem, such as programming language, schema and grammar evolution. Section 3.2.4 surveys work that considers evolution in the context of MDE. Section 3.4 identifies three types of evolution that occur in MDE projects and highlights challenges for their management.

Chapter 4 surveys existing MDE projects and categorises the evolution of MDE development artefacts in those projects. From this survey, the context for the thesis research is narrowed, and the remainder of the thesis focuses on one type of evolution occurring in MDE projects, termed *model-metamodel co-evolution* or simply *co-evolution*. Examples of co-evolution are used to identify the strengths and weaknesses of existing structures and processes for managing co-evolution. From this, Section 4.2.2 identifies a process for managing co-evolution which has not been recognised previously in the literature, Section 4.2.3 derives a categorisation of existing processes for managing co-evolution, and Section 4.3 synthesises requirements for novel structures for managing co-evolution.

Chapter 5 describes novel structures for managing co-evolution, including a metamodel-independent syntax, which is used to identify, report and to facilitate the reconciliation of problems caused by metamodel evolution. The textual modelling notation described in Section 5.2 and the model migration language described in Section 5.4 are used for reconciliation of models in response to metamodel evolution. The latter provides a means for performing reconciliation in a repeatable manner.

Chapter 6 assesses the structures and processes proposed in this thesis by comparison to existing structures and processes. To explore the research hypothesis, several different types of comparison were performed, including an experiment in which quantitive measurements were derived, a collaborative comparison of model migration tools with three MDE experts, and application to a large, independent example of evolution taken from a real-world MDE project.

Chapter 7 summarises the achievements of the research, and discusses results in the context of the research hypothesis. Limitations of the thesis research and areas of future work are also outlined.

# Chapter 2

# Background: Model-Driven Engineering

This chapter presents the thesis background, introduces concepts relating to Model-Driven Engineering (MDE), and surveys the MDE literature. Software evolution research is reviewed in Chapter 3. MDE is a principled approach to software engineering in which models are produced and consumed throughout the engineering process. Section 2.1 introduces the terminology and fundamental principles used in MDE. Section 2.2 reviews guidance and three methods for performing MDE. Section 2.3 describes contemporary MDE environments. Two areas of research relating to MDE, domain-specific languages and language-oriented programming, are discussed in Section 2.4. Finally, the benefits of and current challenges for MDE are described in Section 2.5.

## 2.1 MDE Terminology and Principles

Software engineers using MDE construct and manipulate artefacts familiar from traditional approaches to software engineering (such as code and documentation) and, in addition, work with different types of artefact, such as *models*, *metamodels* and *model transformations*. Furthermore, MDE involves new development activities, such as *model management*. This section describes the artefacts and activities involved in MDE.

### 2.1.1 Models

Models are fundamental to MDE. Kurtev identifies many definitions of the term model [Kurtev 2004], including the following: “any subject using a system A that is neither directly nor indirectly interacting with a system B to obtain information about the system B, is using A as a model for B” [Apostel 1960]. “A model is a representation of a concept. The representation is purposeful and used to abstract from reality the irrelevant details,” [Starfield *et al.* 1990]. “A model is a simplification of a system written in a well-defined language,” [Bézivin & Gerbé 2001].

While there are many definitions of the term model, a common notion is that a model is a representation of the real-world [Kurtev 2004, pg12]. The part of the real-world represented by a model is termed the *domain*, the *object system* or, simply the *system*. Another common notion is that a model may have either a textual or graphical representation [Kolovos *et al.* 2006].

Models that share some characteristics and can be used in place of their object system have been described as *analogous* [Ackoff 1962]. An aeroplane toy that can fly is an analogous model of an aeroplane. In computer science, models can be used to construct a computer system. A model of an object system, say the lending service of a library, might be used to decide the way in which data is stored on disk, or the way in which a program is to be structured.

The models constructed in computer science can be regarded as analogous to two systems: the object system (e.g. the library lending service in the real-world) and the computer system (e.g. the software and hardware used to implement a library lending service) [Jackson 1995]. A model can be used to think about both the real system and the computer system. Figure 2.1 illustrates this notion further and suggests that a model is both the description of the domain (object system) and the machine (computer system) [Jackson 1995]. Computer scientists switch between *designations* when using a model to think about the object system or to think about the software system.

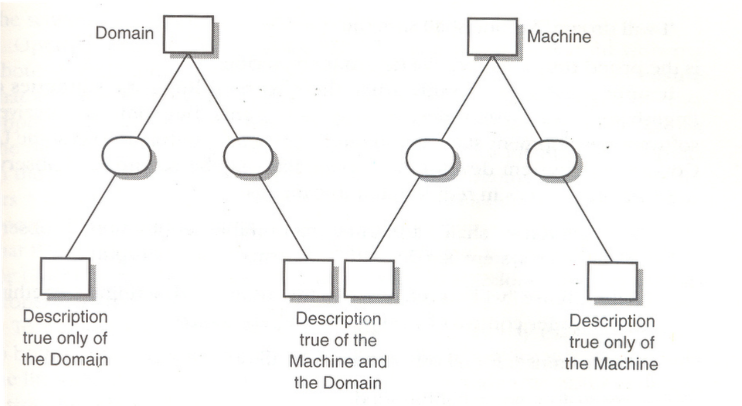


Figure 2.1: Jackson's definition of a model, taken from [Jackson 1995, pg.125].

Models can be unstructured (for example, sketches on a piece of paper) or structured (conform to some well-defined set of syntactic and semantic constraints). In software engineering, models are used widely to reason about object systems and computer systems. MDE recognises this, and seeks to drive the development of computer systems from structured models.

### 2.1.2 Modelling languages

In MDE, models are structured rather than unstructured [Kolovos 2009]. A *modelling language* is the set of syntactic and semantic constraints used to define the structure of a group of related models. In MDE, a modelling language is often specified as a model and, hence the term *metamodel* is used in place of *modelling language*.

*Conformance* is a relationship between a metamodel and a model. A model *conforms to* a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel [Bézivin 2005]. Conformance can be described by a set of constraints between models and metamodels [Paige *et al.* 2007]. When all constraints are satisfied, a model conforms to a metamodel. For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel.

Metamodels facilitate model interchange and, therefore, interoperability between modelling tools. For this reason, Evans recommends that software engineers "use a well-documented shared language that can express the necessary domain information as a common medium of communication." [Evans 2004, pg377]. To support this recommendation, Evans discusses the way in which chemists have collaborated to define a standardised language for describing chemical structures, Chemical Markup Language (CML)[1]. The standardisation of CML has facilitated interoperability between tools for specification, analysis and simulation.

A metamodel typically comprises three categories of constraint:

- **The concrete syntax** provides a notation for constructing models that conform to the language. For example, a model may be represented as a collection of boxes connected by lines. A standardised concrete syntax enables communication. Concrete syntax may be optimised for consumption by machines (e.g. XML Metadata Interchange (XMI) [OMG 2007c]) or by humans (e.g. the Unified Modelling Language (UML) [OMG 2007a]).

- **The abstract syntax** defines the concepts described by the language, such as classes, packages, datatypes. The representation for these concepts is independent of the concrete syntax. For example, the implementation of a compiler might use an abstract syntax tree to encode the abstract syntax of a program (whereas the concrete syntax for the same language may be textual or graphical).

- **The semantics** identifies the meaning of the modelling concepts with respect to the domain. For example, consider a modelling language defined to describe genealogy, and another to describe flora. Although both languages may define a tree construct, the semantics of a tree in one is likely to be different from the semantics of a tree in the other. The semantics of a modelling language may be specified rigorously, by defining a reference semantics in a formal language such as Z [ISO/IEC 2002], or in a semi-formal manner by employing natural language.

---

[1] `http://cml.sourceforge.net/`

Concrete syntax, abstract syntax and semantics are used together to specify modelling languages [Álvarez *et al.* 2001]. There are many other ways of defining languages, but this approach is common in MDE: a metamodel is often used to define abstract syntax, a grammar or text-to-model transformation to specify concrete syntax, and code generators, annotated grammars or behavioural models to effect semantics.

### 2.1.3 MOF: A metamodelling language

Software engineers using MDE can re-use existing – and define new – metamodels. To facilitate interoperability between MDE tools, the Object Management Group (OMG)[2] has standardised a language for specifying metamodels, the Meta-Object Facility (MOF). Contemporary MDE tools are interoperable because MOF standardises the way in which models are represented with a further OMG standard, XML Metadata Interchange (XMI), a dialect of XML optimised for loading, storing and exchanging models.

Because MOF is a modelling language for describing modelling languages, it is sometimes termed a metamodelling language. Part of the UML metamodel, defined in MOF, is shown in Figure 2.2. As discussed in Section 2.3, different kinds of concrete syntax can be used for MOF. Figure 2.2, for example, uses a concrete syntax similar to that of UML class diagrams. Specifically:

- Modelling constructs are drawn as boxes. The name of each modelling construct is emboldened. The name of abstract (uninstantiable) constructs are italicised.
- Attributes are contained within the box of their modelling construct. Each attribute has a name, a type (prefixed with a colon) and may define a default value (prefixed with an equals sign).
- Generalisation is represented using a line with an open arrow-head.
- References are specified using a line. An arrow illustrates the direction in which the reference may be traversed. Labels are used to name and define the multiplicity of references.
- Containment references are specified by including a solid diamond on the containing end.

---

[2] `http://www.omg.org`

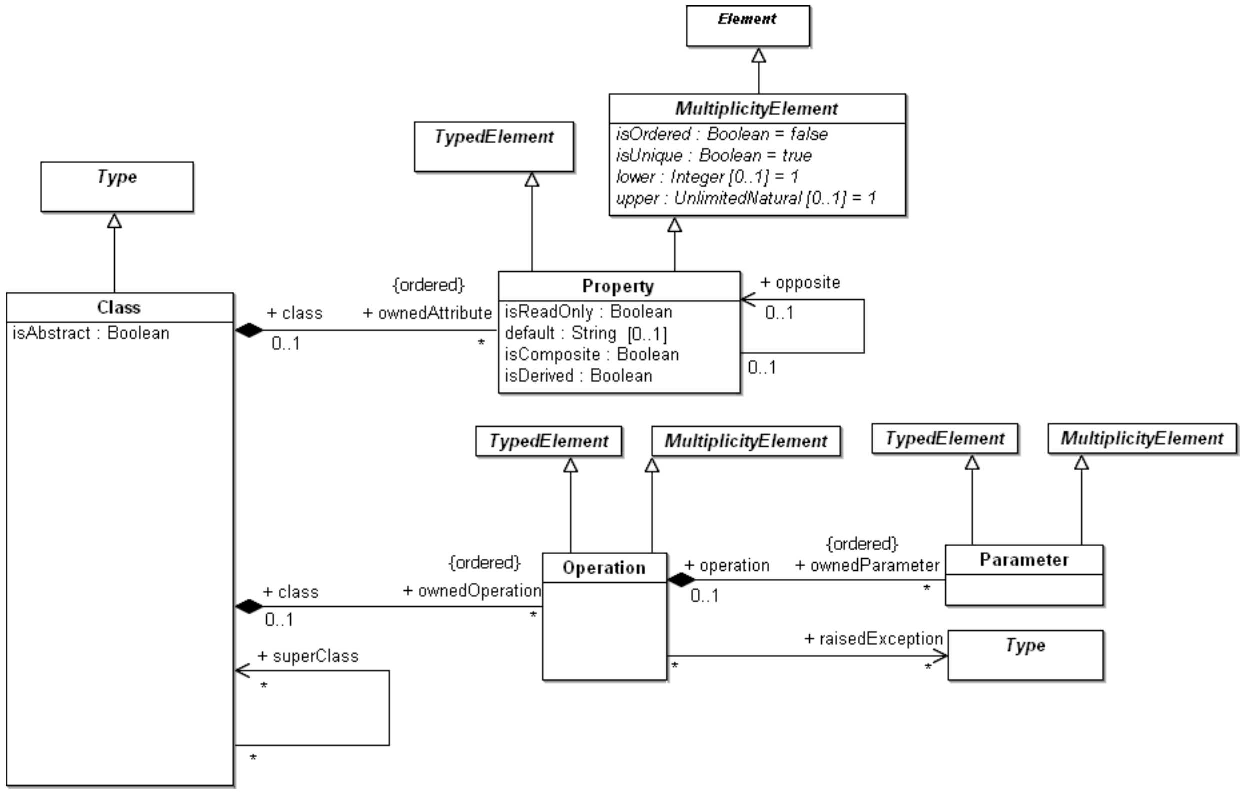


Figure 2.2: A fragment of the UML metamodel defined in MOF, from [OMG 2007a].

Specifying modelling languages with a common metamodelling language, such as MOF, ensures consistency in the way in which modelling constructs are specified. MOF has facilitated the construction of interoperable MDE tools that can be used with a range of modelling languages. Without a standardised metamodelling language, modelling tools were specific to one modelling language, such as UML. In contemporary MDE environments, any number of modelling languages can be used together and manipulated in a uniform manner.

Furthermore, when modelling languages are specified without using a common metamodelling language, identifying similarities between modelling languages is challenging [Frankel 2002, pg97]. The sequel discusses the way in which models and metamodels are used to construct systems in MDE.

### 2.1.4 Model Management

In MDE, models are *managed* to produce software. The term *model management* was first used in 2004 to describe a collection of operators for manipulating models [Melnik 2004]. This thesis uses the term *model management* to refer to development activities that manipulate models for the purpose of producing software. Model management activities typical in MDE, such as model transformation and validation, are discussed in this section. Section 2.2 discusses MDE guidelines and methods, and describes the way in which model management activities are used together to produce software in MDE.

#### Model Transformation

Model transformation is a development activity in which software artefacts are derived from others, according to some well-defined specification. There are three types of model transformations: those specified between modelling languages (model-to-model transformation), those specified between modelling languages and textual artefacts (model-to-text-transformation) and those specified between textual artefacts and modelling languages (text-to-model transformation) [Kleppe *et al.* 2003]. Each type of transformation has unique characteristics and tools, but share some common characteristics. The remainder of this section first introduces the commonalities and then discusses each type of transformation individually.

**Common characteristics of model transformations** The input to a transformation is termed its *source*, and the output its *target*. In theory, a transformation can have more than one source and more than one target, but not all transformation languages support multiple sources and targets. Consequently, much of the model transformation literature considers single source and target transformations.

The similarities and differences of different model transformation languages have been categorised and compared using a feature model [Czarnecki & Helsen 2006]. Two features peculiar to model transformation are relevant to the research presented in this thesis, and are now discussed.

**Source-target relationship** A *new-target* transformation creates afresh the target model on each invocation, while an *existing-target* transformation updates an existing model. Existing target transformations are used for partial (incremental) transformation and for preserving parts of the target model that are not derived from the source model.

**Domain language** Transformations specified between source and target models that conform to the same metamodel are termed *endogenous* or *rephrasings*, while transformations specified between a source and a target model that conform to different metamodels are termed *exogenous* or *translations*.

Endogenous, existing-target transformations are a special case of transformation and are termed *refactorings*. Refactorings have been studied in the context of software evolution and are discussed more thoroughly in Chapter 3.

**Model-to-Model (M2M) Transformation** M2M transformation is used to derive models from others. By automating the derivation of models from others, M2M transformation has the potential to reduce the cost of engineering large and complex systems that can be represented as a set of interdependent models [Sendall & Kozaczynski 2003].

M2M transformations are most often specified using a set of *transformation rules* [Czarnecki & Helsen 2006]. Each rule specifies the way in which a specific set of elements in the source model is transformed to an equivalent set of elements in the target model [Kolovos 2009, pg.44].

Many M2M transformation languages have been proposed, such as the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005], VIsual Automated model TRAnsformations (VIATRA) [Varró & Balogh 2007] and the Epsilon Transformation Language (ETL) [Kolovos *et al.* 2008b]. There also exists a standard for M2M transformation, Queries/Views/Transformations (QVT) [OMG 2005]. M2M transformation languages can be categorised according to their *style*, which is either declarative, imperative or hybrid.

Declarative M2M transformation languages only provide constructs for mapping source to target model elements and, as such, are not computationally complete. Consequently, the scheduling of rules can be *implicit* (determined by the execution engine of the transformation language). By contrast, imperative M2M transformation languages are computationally complete, but often require rule scheduling to be *explicit* (specified by the user). Hybrid M2M transformation languages combine declarative and imperative parts, are computationally complete, and provide a mixture of implicit and explicit rule scheduling.

Declarative M2M transformation languages cannot be used to solve some categories of transformation problem [Patrascoiu & Rodgers 2004]. Imperative M2M transformation languages are argued to be difficult to write and maintain [Kolovos 2009, pg.45]. Consequently, hybrid languages, such as ATL, are presently believed to be more suitable for specifying model transformation than pure imperative or declarative languages [Kolovos *et al.* 2008b].

An example of an M2M transformation, written in the hybrid M2M transformation language ETL, is shown in Listing 2.1. The source of the transformation is a state machine model, conforming to the metamodel shown in Figure 2.3. The target of the transformation is an object-oriented model, conforming to the metamodel shown in Figure 2.4. The transformation in Listing 2.1 comprises two rules.

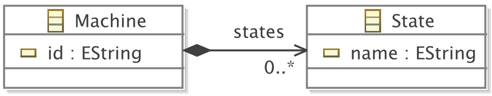


Figure 2.3: A State Machine metamodel.

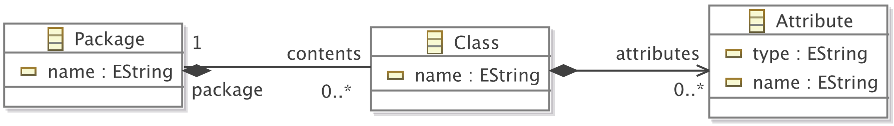


Figure 2.4: An Object-Oriented metamodel.

The first rule (lines 1-7) is named `Machine2Package` (line 1) and transforms *Machine*s (line 2) into *Package*s (line 3). The body of the first rule (lines 5-6) specifies the way in which a `Package`, `p`, can be derived from a `Machine`, `m`. Specifically, the `name` of `p` is derived from the `id` of `m` (line 5), and the `contents` of `p` are derived from the `states` of `m` (line 6).

The second rule (lines 9-16) transforms `States` (line 10) to `Classes` (line 11). Additionally, line 13 contains a *guard* to specify that the rule is only to be applied to `States` whose `isFinal` property is `false`.

When executed, the transformation rules will be scheduled **implicitly** by the execution engine, and invoked once for each `Machine` and `State` in the source. On line 6 of Listing 2.1, the built-in `equivalent()` operation is used to produce a set of `Classes` from a set of `States` by invoking the relevant transformation rule. This is an example of **explicit** rule scheduling, in which the user defines when a rule will be called.

```
1  rule Machine2Package
2    transform m : StateMachine!Machine
3    to       p : ObjectOriented!Package {
4
5    p.name    := 'uk.ac.york.cs.' + m.id;
6    p.contents := m.states.equivalent();
7  }
```

```

rule State2Class
  transform s : StateMachine!State
  to        c : ObjectOriented!Class

  guard: not s.isFinal {

  c.name := s.name + 'State';
}
```

Listing 2.1: M2M transformation in the Epsilon Transformation Language [Kolovos *et al.* 2008b]

**Model-to-Text (M2T) Transformation** M2T transformation is used for model serialisation (enabling model interchange), code and documentation generation, and model visualisation and exploration. In 2005, the OMG recognised the lack of a standardised M2T transformation with its M2T Language Request for Proposals[3]. In response, various M2T languages were developed including JET[4], XPand[5], MOFScript [Oldevik *et al.* 2005], the Epsilon Generation Language (EGL) [Rose *et al.* 2008b].

Because M2T transformation is used to produce unstructured rather than structured artefacts, M2T transformation has different requirements to M2M transformation. For instance, M2T transformation languages often provide mechanisms for specifying sections of text that will be completed manually and must not be overwritten by the transformation engine.

*Template*s are commonly used in M2T languages. Templates comprise *static* and *dynamic* sections. When the transformation is invoked, the contents of static sections are emitted verbatim, while dynamic sections contain logic and are executed.

An exemplar M2T transformation, written in EGL, is shown in Listing 2.2. The source of the transformation is an object-oriented model conforming to the metamodel shown in Figure 2.4, and the target is Java source code. The template assumes that an instance of `Class` is stored in the `class` variable.

```
package [%=class.package.name%];

public class [%=class.name%] {
  [% for(attribute in class.attributes) { %]
    private [%=attribute.type%] [%=attribute.name%];
  [% } %]
}
```

Listing 2.2: M2T transformation in the Epsilon Generation Language [Rose *et al.* 2008b]

---

[3] `http://www.omg.org/docs/ad/04-04-07.pdf`

[4] `http://www.eclipse.org/modeling/m2t/?project=jet`

[5] `http://www.eclipse.org/modeling/m2t/?project=xpand`

In EGL, dynamic sections are contained within `[%` and `%]`. *Dynamic output* sections are a specialisation of dynamic sections contained within `[%=` and `%]`. The result of evaluating a dynamic output section is included in the generated text. Line 1 of Listing 2.2 contains two static sections (‘`package`’ and ‘`;`’) and a dynamic output section (`[%=class.package.name%]`), and will generate a package declaration when executed. Similarly, line 3 will generate a class declaration. Lines 4 to 6 iterate over every `attribute` of the `class`, outputting a field declaration for each `attribute`.

**Text-to-Model (T2M) Transformation** T2M transformation is most often implemented as a parser that generates a model rather than object code. Parser generators such as ANTLR [Parr 2007] can be used to produce a structured artefact (such as an abstract syntax tree) from text. T2M tools typically reuse a parser generator, and post-process the structured artefacts to produce a model that can be managed with a particular modelling framework.

Xtext[6] and EMFText [Heidenreich *et al.* 2009] are contemporary examples of T2M tools that, given a grammar and a target metamodel, will automatically generate a parser that transforms text to a model.

An exemplar T2M transformation, written in EMFText, is shown in Listing 2.3. From the transformation shown in Listing 2.3, EMFText can be used to generate a parser that, when executed, will produce state machine models. For the input, `lift[stationary up down stopping emergency]`, the parser will produce a model containing one `Machine` with `lift` as its `id`, and five `States` with the `names`, `stationary`, `up`, `down`, `stopping`, and `emergency`.

Lines 1-2 of Listing 2.3 define the name of the parser and target metamodel. Line 3 indicates that parser should first seek to construct a `Machine` from the source text. Lines 5-9 define rules for the lexer, including a rule for recognising `IDENTIFIERs` (represented as alphabetic characters).

Lines 11-14 of Listing 2.3 are key to the transformation. Line 11 specifies that a `Machine` is constructed whenever an `IDENTIFIER` is followed by a `LBRACKET` and eventually a `RBRACKET`. When constructing a `Machine`, the first time an `IDENTIFIER` is encountered, it is stored in the `id` attribute of the `Machine`. The `states*` statement on line 12 indicates that, before matching a `RBRACKET`, the parser is permitted to transform subsequent text to a `State` (according to the rule on line 13) and store the resulting `State` in the `states` reference of the `Machine`. The asterisks in `states*` indicates that any number of `States` can be constructed and stored in the `states` reference.

```
1  SYNTAXDEF statemachine
2  FOR <statemachine>
3  START Machine
4
5  TOKENS {
6    DEFINE IDENTIFIER $('a'..'z'|'A'..'Z')*$;
```

---

[6] `http://www.eclipse.org/Xtext/`

```
  DEFINE LBRACKET $'['$;
  DEFINE RBRACKET $']'$;
}

RULES {
  Machine ::= id[IDENTIFIER] LBRACKET states* RBRACKET ;
  State ::= name[IDENTIFIER] ;
}
```

Listing 2.3: T2M transformation in EMFText

### Model Validation

Model validation provides a mechanism for managing the integrity of the software developed using MDE. A model that omits information is said to be *incomplete*, while related models that suggest differences in the underlying phenomena are said to be *contradicting* [Kolovos 2009]. Incompleteness and contradiction are two examples of *inconsistency* [Elaasar & Briand 2004]. In MDE, inconsistency is detrimental, because, when artefacts are automatically derived from each other, the inconsistency of one artefact might be propagated to others. Model validation is used to detect, report and reconcile inconsistency throughout a MDE process.

Inconsistency detection is inherently pattern-based and, hence, higher-order languages are more suitable for model validation than so-called "third-generation" programming languages (such as Java) [Kolovos 2009]. The Object Constraint Language (OCL) [OMG 2006] is an OMG standard that can be used to specify consistency constraints on UML and MOF models. OCL cannot express inter-model constraints, unlike the Epsilon Validation Language (EVL) [Kolovos *et al.* 2009] and the xlinkit toolkit [Nentwich *et al.* 2003].

An exemplar model validation constraint, written in EVL, is shown in Listing 2.4. The constraint validates state machine models that conform to the metamodel shown in Figure 2.3. The constraint shown in Listing 2.4 is defined for `States` (line 1), and checks that there exists some transition whose source or target is the current state (line 4). When the check part (line 4) is not satisfied, the message part (line 6) is displayed. When executed, the EVL constraint will be invoked once for every `State` in the model. The keyword `self` is used to refer to the particular `State` on which the constraint is currently being invoked.

```
1 context State {
2   constraint NoStateIsAnIsland {
3     check:
4       Transition.all.exists(t | t.source == self or t.target == self)
5     message:
6       'The state ' + self.name + ' has no transitions.'
7   }
```

```
}
```

Listing 2.4: Model validation in the Epsilon Validation Language

#### Further model management activities

In addition to model transformation and validation, further examples of model management activities include model merging or weaving – in which two or more models are combined (e.g. Reuseware [Henriksson *et al.* 2008]) – and model comparison in which a *trace* of similar and different elements is produced from two or more models (e.g. EMF Compare [Brun & Pierantonio 2008]).

Further activities, such as model versioning and tracing, might be regarded as model management but, in the context of this thesis, are considered as evolutionary activities and as such are discussed in Chapter 3.

### 2.1.5 Summary

This section has introduced the terminology and principles necessary for discussing MDE in this thesis. Models provide abstraction, capturing necessary and disregarding irrelevant details. Metamodels provide a structured mechanism for describing the syntactic and semantic rules to which a model must conform. Metamodels facilitate interoperability between modelling tools. MOF, the OMG standard metamodelling language, enables the development of tools that can be used with a range of metamodels, such as model management tools. Throughout an MDE process, models are manipulated to produce other development artefacts using model management activities such as model transformation and validation. Using the terms and principles described in this section, the ways in which MDE is performed in practice are now discussed.

## 2.2 MDE Guidelines and Methods

For performing MDE, new engineering practices and processes have been proposed. Proponents of MDE have produced guidance and methods for MDE. This section discusses the guidance for MDE set out in the Model-Driven Architecture [OMG 2008b] and three popular methods for MDE.

### 2.2.1 The Model-Driven Architecture

The Model-Driven Architecture (MDA) is a software engineering framework defined by the OMG. The MDA provides guidelines for MDE. For instance, the MDA prescribes the use of a Platform Independent Model (PIM) and one or more Platform Specific Models (PSMs).

A PIM provides an abstract, implementation-agnostic view of a system. Successive PSMs provide increasingly more implementation detail. Inter-model mappings are used to forward- and reverse-engineer these models, as depicted in Figure 2.5.

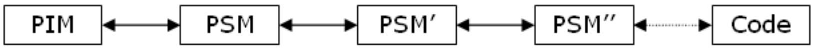


Figure 2.5: Interactions between a PIM and several PSMs.

A key difference between the MDA and related approaches, such as round-trip engineering (in which models and code are co-evolved to develop a system), is that the MDA prescribes automated transformations between PIM and PSMs, whereas other approaches use some manual transformations.

With MDA, the OMG sought to communicate and encourage the adoption of MDE principles, and to provide standards for building interoperable MDE platforms. Arguably, some parts of MDA have been widely adopted. For example, the metamodelling language provided by the Eclipse Modeling Framework (Section 2.3.1) is heavily based on MOF (Section 2.1.3), one of the key standards prescribed by MDA. However, it is difficult to assess the extent to which the principles advocated by MDA have been adopted. Empirical analysis is needed to determine the way in which MDE is performed in practice, and to drive changes to MDA and the modelling standards provided by the OMG.

### Standards for the MDA

As part of the guidelines for MDE, the OMG prescribes a set of standards for the MDA. The standards are allocated to one of four tiers, and each tier represents a different level of model abstraction (Figure 2.6). Members of one tier conform to a member of the tier above. A discussion of the four tiers, based on [Kleppe *et al.* 2003, Section 8.2], follows.

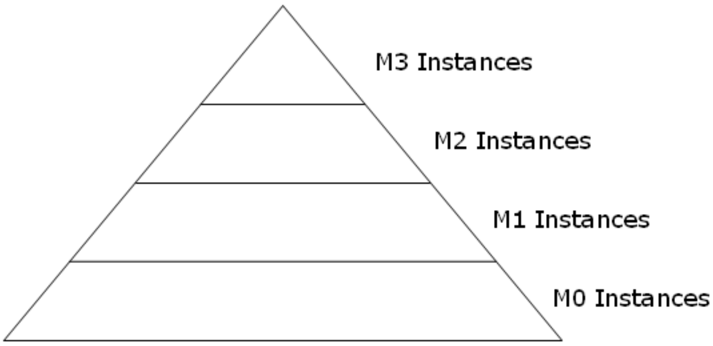


Figure 2.6: The tiers of standards used as part of MDA.

The base of the pyramid, tier M0, contains the domain (real-world). When modelling a business, this tier is used to describe items of the business itself, such as a real customer or an invoice. When modelling software, M0 instances describe the software representation of such items. M1 contains models (Section 2.1.1) of the concepts in M0, for example a customer may be represented as a class with attributes. The M2 tier contains the modelling languages (metamodels, Section 2.1.2) used to describe the contents of the M1 tier. For example, if UML [OMG 2007a] models were used to describe concepts as classes in the M1 tier, M2 would contain the UML metamodel. Finally, M3 contains a metamodelling language (metametamodel, Section 2.1.3) which describes the modelling languages in the M2 tier. As discussed in Section 2.1.3, the M3 tier facilitates tool standardisation and interoperability. The MDA specifies the Meta-Object Facility (MOF) [OMG 2008a] as the only member of the M3 tier.

### Interpretations of the MDA

The guidelines in the MDA have been interpreted by engineers in two distinct ways [McNeile 2003]. Both interpretations begin with a PIM, but the way in which executable code is produced differs:

- **Translationist**: The PIM is used to generate code directly using a sophisticated code generator. Any intermediate PSMs are internal to the code generator. No generated artefacts are edited manually.

- **Elaborationist**: Any generated artefacts (such as PSMs, code and documentation) can be augmented with further details of the application. To ensure that all models and code are synchronised, tools must allow bi-directional transformations.

Translationists must encode behaviour in PIMs [Mellor & Balcer 2002], whereas elaborationists can specify behaviour in PSMs or in code [Kleppe *et al.* 2003].

The difference between translationist and elaborationist approaches to MDE is related to a difference in the way in which models are viewed in traditional approaches to software engineering. Modelling has been identified as a vital activity that should take place throughout the development process [Evans 2004]. By contrast, it has also been suggested that modelling should only be used for communicating and reasoning about a design, and not “as a long-term replacement for real, working software” [Martin & Martin 2006, ch. 14].

### Use of MDA

The guidelines set out in the MDA have been adopted – in varying degrees – by the methods for MDE described in the sequel. Tools for MDE (Section 2.3) tend not to enforce many of the MDA guidelines. For example, contemporary modelling frameworks allow – but do not force – developers to specify a single PIM and use model transformation to create successive PSMs. The MDE standards prescribed by the MDA have been widely adopted. MOF, UML and XMI, for example, are used in many contemporary modelling frameworks and modelling tools.

### 2.2.2 Methods for MDE

Several methods for MDE are prevalent today. In this section, three of the most established MDE methods are discussed: Architecture-Centric Model-Driven Software Development [Stahl *et al.* 2006], Domain-Specific Modelling [Kelly & Tolvanen 2008] and Software Factories [Greenfield *et al.* 2004] at Microsoft. All three methods have been defined by MDE practitioners, and have been used repeatedly to solve problems in industry. The methods vary in the extent to which they follow the guidelines set out by the MDA.

#### Architecture-Centric Model-Driven Software Development

*Architecture-centric model-driven software development* (AC-MDSD) is a style of MDE that focuses on generating the infrastructure of large applications [Stahl *et al.* 2006]. For example, a typical J2EE application contains concepts (such as EJBs, descriptors, home and remote interfaces) that "admittedly contain domain-related information such as method signatures, but which also exhibit a high degree of redundancy" [Stahl *et al.* 2006]. Using code generation, AC-MDSD seeks to eliminate this kind of redundancy. Domain-related information is specified in a single source (typically in a model). The single source is used as input to code generators, which automatically produce the implementation concepts.

AC-MDSD applies more of the MDA guidelines than the other methods discussed below. For instance, AC-MDSD supports the use of a general-purpose modelling language for specifying models. The way in which AC-MDSD may be used to enhance the productivity, efficiency and understandability of software development has been demonstrated for a small number of examples [Stahl *et al.* 2006]. In these examples, models are annotated using UML profiles to describe domain-specific concepts.

#### Domain-Specific Modelling

Domain-Specific Modelling (DSM) [Kelly & Tolvanen 2008] provides a collection of principles, practices and advice for constructing systems using MDE. DSM is based on the translationist interpretation of the MDA: domain models are transformed directly to code. DSM has been motivated by comparison to the large productivity gains made when third-generation programming languages were used in place of assembler [Kelly & Tolvanen 2008]. Tolvanen[7] notes that DSM focuses on increasing the productivity of software engineering by allowing developers to specify solutions by using models that describe the application domain.

To perform DSM, expert developers define:

- **A domain-specific modelling language**: allowing domain experts to encode solutions to their problems.

---

[7] Tutorial on Domain Specific Modelling for Full Code Generation at the Fourth European Conference on Model Driven Architecture (ECMDA), June 2008, Berlin, Germany.

- **A code generator**: that translates the domain-specific models to executable code in an existing programming language.
- **Framework code**: that encapsulates the common areas of all applications in this domain.

As the development of these three artefacts requires significant effort from expert developers, Tolvanen[7] states that DSM should only be applied if more than three problems specific to the same domain are to be solved.

Tools for defining domain-specific modelling languages, editors and code generators enable DSM [Kelly & Tolvanen 2008]. Reducing the effort required to specify these artefacts is key to the success of DSM. In this respect, DSM resembles a programming paradigm termed *language-oriented programming* (LOP), which also requires tools to simplify the specification of new languages. LOP is discussed further in Section 2.4.

Examples from industrial partners have been used to argue that DSM can improve developer productivity [Kelly & Tolvanen 2008]. Unlike the MDA, DSM appears to be optimised for increasing productivity, and less concerned with portability or maintainability. Therefore, DSM is less suitable for engineering applications that frequently interoperate with – and are underpinned by – changing technologies.

### Microsoft Software Factories

During the industrialisation of the automobile industry, problems with economies of scale (mass production) and scope (product variation) were largely addressed and, as a consequence, the efficiency of automobile production increased [Greenfield *et al.* 2004, pg159]. Software Factories [Greenfield *et al.* 2004], a software engineering method developed at Microsoft, seeks to address problems with economies of scope in software engineering by borrowing concepts from product-line engineering. Software Factories focuses on addressing economies of scope rather than economies of scale because, unlike many other engineering disciplines, software development requires considerably more development effort than production effort [Greenfield *et al.* 2004].

Software Factories [Greenfield *et al.* 2004] prescribes a bottom-up approach to abstraction and re-use. Development begins by producing prototypical applications. The common elements of these applications are identified and abstracted into a product-line. When instantiating a product, models are used to choose values for the variation points in the product. To simplify the creation of these models, Software Factories propose model creation wizards, because "moving from totally-open ended hand-coding to more constrained forms of specification [such as wizard-based feature selection] are the key to accelerating software development" [Greenfield *et al.* 2004, pg179]. By providing explanations that assist in making decisions, the wizards used in Software Factories guide users towards best practices for customising a product.

Compared to DSM, Software Factories appears to provide more support for addressing portability problems. The latter provides *viewpoints* into the product-line (essentially different ways of presenting and aggregating data from development artefacts), which allow decoupling of concerns (e.g. between logical, conceptual and physical layers). Viewpoints provide a mechanism for abstracting over different layers of platform independence, adhering more closely than DSM to the guidelines provided in the MDA. Unlike the guidelines provided in the MDA, Software Factories does not insist that development artefacts be derived automatically where possible.

Finally, Software Factories prescribes the use of domain-specific languages (discussed in Section 2.4.1) for describing models in conjunction with Software Factories, rather than general-purpose modelling languages, as the authors of Software Factories state that the latter often have imprecise semantics [Greenfield *et al.* 2004].

### 2.2.3 Summary

This section has discussed the ways in which process and practices for MDE have been captured. Guidance for MDE has been set out in the MDA standard, which seeks to use MDE to produce adaptable software in a productive and maintainable manner. Three methods for performing MDE have been discussed.

The methods discussed share some characteristics. They all require a set of exemplar applications, which are examined by MDE experts. Analysis of the exemplar applications identifies the way in which software development may be decomposed. A modelling language for the problem domain is constructed, and instances are used to generate future applications. Code common to all applications in the problem domain is encapsulated in a framework.

Each method has a different focus. AC-MDSD seeks to eliminate duplication of information from the problem domain via automatic code generation, and targets enterprise applications. Software Factories concentrates on providing different viewpoints into the system, and facilitating collaborative specification of a system. DSM aims to improve reusability between solutions to problems in the same problem domain, and hence improve developer productivity.

Perhaps unsurprisingly, the proponents of each method for MDE recommend a single tool (such as MetaCase for DSM). Alternative tools are available from open-source modelling communities, including the Eclipse Modelling Project, which provides – among other MDE tools – arguably the most widely used MDE modelling framework. Two MDE tools are reviewed in the sequel.

## 2.3 Tools for MDE

Mature and powerful tools and languages for many common MDE activities are available today. This section discusses two MDE tools that are well-suited for MDE research and that are used in the remainder of the thesis.

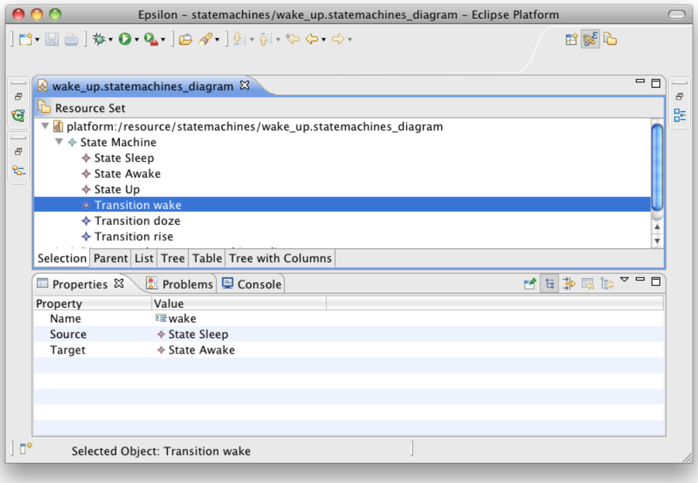
Section 2.3.1 provides an overview of the Eclipse Modelling Framework (EMF) [Steinberg *et al.* 2008], which implements MOF and underpins many contemporary MDE tools and languages, facilitating their interoperability. Section 2.3.2 discusses Epsilon [Kolovos 2009], an extensible platform for the specification of model management languages. The highly extensible nature of Epsilon (which is described below) makes it an ideal host for the rapid prototyping of languages and exploring research hypotheses.

The purpose of this section is to review EMF and Epsilon, and not to provide a thorough review of all MDE tools. There are many other MDE tools and environments that this section does not discuss, such as ATL and VIATRA for M2M transformation (Section 2.1.4), oAW[8] for model transformation and validation, and the AMMA platform[9] for large-scale modelling, model weaving and software modernisation.

### 2.3.1 Eclipse Modelling Framework

The Eclipse Foundation[10] is an open-source community seeking to build an extensible and integrated development platform. Within Eclipse, the Eclipse Modelling Framework (EMF) project [Steinberg *et al.* 2008] provides support for MDE. EMF provides code generation facilities, and a meta-modelling language, Ecore, that implements the MOF 2.0 standard [OMG 2008a]. EMF is arguably the most widely-used contemporary MDE modelling framework.

EMF is used to generate metamodel-specific editors for loading, storing and constructing models. EMF model editors comprise a navigation view for specifying the elements of the model, and a properties view for specifying the features of model elements. Figure 2.7 shows an EMF model editor for a simple state machine language. The navigation (or tree) view is shown in the top pane, while the properties view is shown in the bottom pane.



- Resource Set
  - platform:/resource/statemachines/wake_up.statemachines_diagram
    - State Machine
      - State Sleep
      - State Awake
      - State Up
      - Transition wake
      - Transition doze
      - Transition rise

| Property | Value |
|---|---|
| Name | wake |
| Source | State Sleep |
| Target | State Awake |

Selected Object: Transition wake

Figure 2.7: An EMF model editor for state machines.

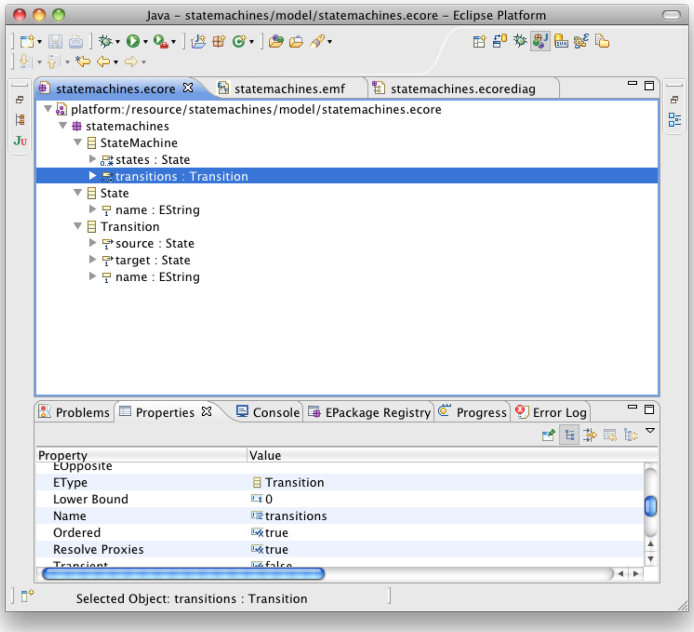
---

[8] http://www.eclipse.org/workinggroups/oaw/

[9] http://wiki.eclipse.org/AMMA

[10] http://www.eclipse.org

Users of EMF can define their own metamodels in Ecore, the metamodelling language and MOF implementation of EMF. EMF provides two metamodel editors, tree-based and graphical. Figure 2.8 shows the metamodel of a simple state machine language in the tree-based metamodel editor. Figure 2.9 shows the same metamodel in the graphical metamodel editor. Like MOF, the graphical metamodel editor uses concrete syntax similar to that of UML class diagrams. Emfatic[11] provides a further, textual metamodel editor for EMF, and is shown in Figure 2.10. The editors shown in Figure 2.8, 2.9 and 2.10 are used to manipulate the same underlying metamodel, but using different syntaxes. A change to the metamodel in one editor can be propagated automatically to the other two.



```
▼ platform:/resource/statemachines/model/statemachines.ecore
  ▼ statemachines
    ▼ StateMachine
      ▶ states : State
      ▶ transitions : Transition
    ▼ State
      ▶ name : EString
    ▼ Transition
      ▶ source : State
      ▶ target : State
      ▶ name : EString
```

| Property | Value |
|---|---|
| EOpposite | |
| EType | Transition |
| Lower Bound | 0 |
| Name | transitions |
| Ordered | true |
| Resolve Proxies | true |
| Transient | false |

Selected Object: transitions : Transition

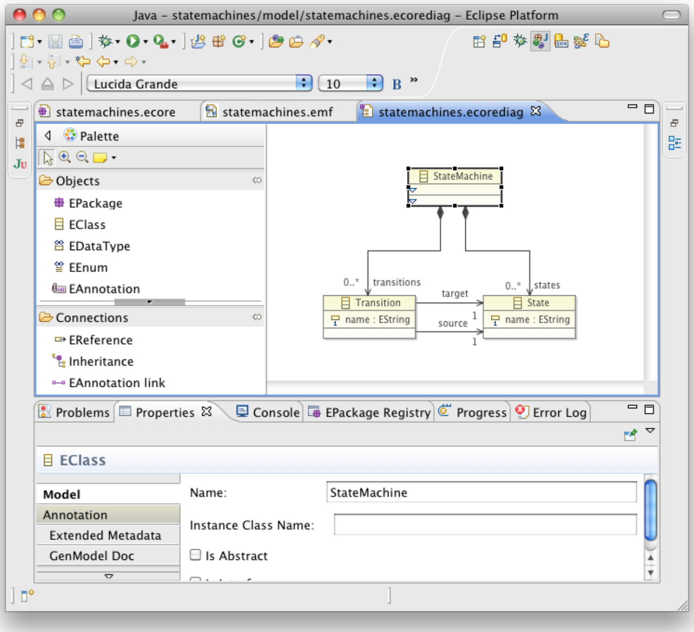
Figure 2.8: EMF's tree-based metamodel editor.

From a metamodel, EMF can generate an editor for models that conform to that metamodel. For example, the simple state machine metamodel specified in Figures 2.8, 2.9 and 2.10 was used to generate the code for the model editor shown in Figure 2.7. The model editors generated by EMF incorporate mechanisms for loading and saving models. As prescribed by MOF, EMF typically generates code that stores models using XMI [OMG 2007c], a dialect of XML optimised for model interchange.

The Graphical Modeling Framework (GMF) [Gronback 2009] is used to create graphical model editors from metamodels defined with EMF. Figure 2.11 shows a model editor produced with GMF for the simple state machine language described

---

[11] `http://www.alphaworks.ibm.com/tech/emfatic`



Name: StateMachine

Instance Class Name:

☐ Is Abstract

Figure 2.9: EMF's graphical metamodel editor.

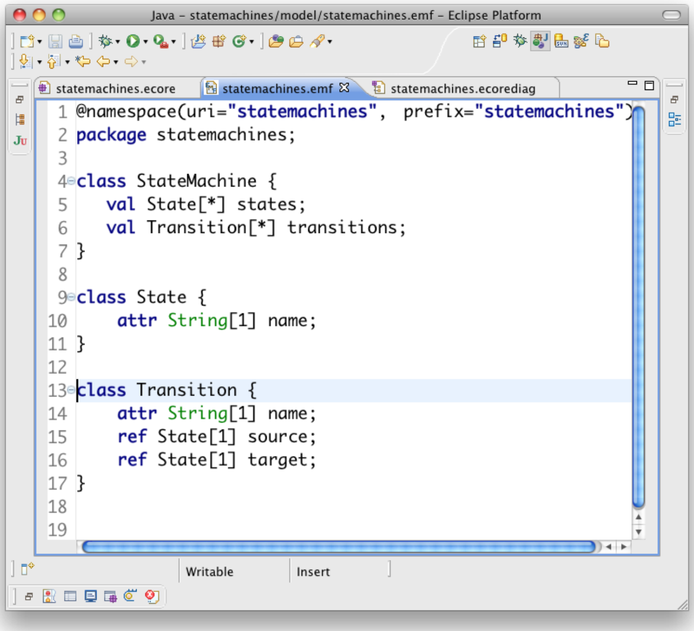
above. GMF itself uses a model-driven approach: users specify several models, which are combined, transformed and then used to generate code for the resulting graphical editor.

Many MDE tools are interoperable with EMF, enriching its functionality. The remainder of this section discusses one tool that is interoperable with EMF, Epsilon.

### 2.3.2 Epsilon

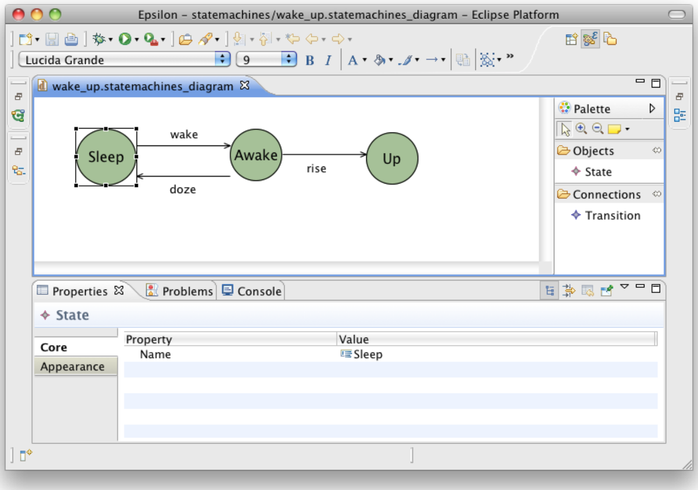
The Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [Kolovos 2009] is a suite of tools and domain-specific languages for MDE. Epsilon comprises several integrated model management languages – built on a common infrastructure – for performing tasks such as model transformation, model validation and model merging [Kolovos 2009]. Figure 2.12 illustrates the various components of Epsilon in 2008, at the start of the thesis research. Since then, several further languages and tools have been added to Epsilon, including those presented in Chapter 5.

Whilst many model management languages are bound to a particular subset of modelling technologies, limiting their applicability, Epsilon's model management languages can manipulate models written in various modelling languages [Kolovos *et al.* 2006]. Currently, Epsilon supports models implemented using EMF, MOF 1.4, XML, or Com-



```
@namespace(uri="statemachines", prefix="statemachines")
package statemachines;

class StateMachine {
   val State[*] states;
   val Transition[*] transitions;
}

class State {
     attr String[1] name;
}

class Transition {
     attr String[1] name;
     ref State[1] source;
     ref State[1] target;
}
```

Figure 2.10: The Emfatic textual metamodel editor for EMF.



| Property | Value |
|---|---|
| Name | Sleep |

Figure 2.11: GMF state machine model editor.

munity Z Tools (CZT)[12]. Interoperability with further modelling technologies can be achieved by extension of the Epsilon Model Connectivity (EMC) layer.

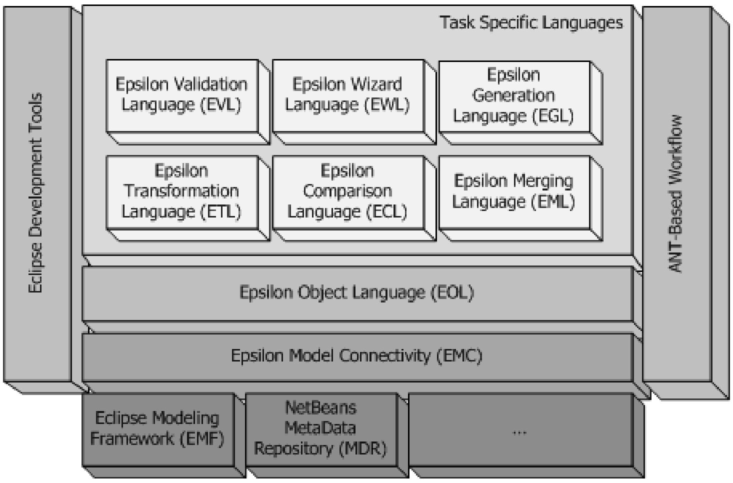


Figure 2.12: The architecture of Epsilon, taken from [Rose *et al.* 2008b].

The architecture of Epsilon promotes reuse when building task-specific model management languages and tools. Each Epsilon language can be reused wholesale in the production of new languages. Ideally, the developer of a new language only has to design language concepts and logic that do not already exist in Epsilon languages. As such, new task-specific languages can be implemented in a minimalistic fashion. This claim has been demonstrated by the style of implementation used to construct the Epsilon Generation Language (EGL) [Rose *et al.* 2008b].

The Epsilon Object Language (EOL) [Kolovos *et al.* 2006] is the core of the platform and provides functionality similar to that of OCL [OMG 2006]. However, EOL provides an extended feature set, which includes the ability to update models, access to multiple models, conditional and loop statements, statement sequencing, and provision of standard output and error streams.

As shown in Figure 2.12, every Epsilon language re-uses EOL, so improvements to EOL enhance the entire platform. EOL also allows developers to delegate computationally intensive tasks to extension points, where the task can be authored in Java.

Epsilon is a member of the Eclipse GMT[13] project, a research incubator for the top-level modelling technology project. Epsilon provides a lightweight means for defining new experimental languages for MDE. For these reasons, Epsilon is uniquely positioned as an ideal host for the rapid prototyping of languages for model management, and hence has been used extensively for the work described in Chapter 5.

---

[12] `http://czt.sourceforge.net/`

[13] `http://www.eclipse.org/gmt`

### 2.3.3 Summary

This section has introduced the MDE tools used throughout the remainder of the thesis. The Eclipse Modeling Framework (EMF) provides an implementation of MOF, Ecore, for defining metamodels. From metamodels defined in Ecore, EMF can generate code for metamodel-specific editors and for persisting models to disk. EMF is arguably the most widely used contemporary MDE modelling framework and its functionality is enhanced by numerous tools, such as the Graphical Modeling Framework (GMF) and Epsilon. GMF allows metamodel developers to specify a graphical concrete syntax for metamodels, and can be used to generate graphical model editors. Epsilon is an extensible platform for defining and executing model management languages, provides a high degree of re-use for defining new model management languages and can be used with a range of modelling frameworks, including EMF.

## 2.4 Research Relating to MDE

MDE is closely related to several other fields of software engineering. This section discusses two of those fields, Domain-Specific Languages (DSLs) and Language-Oriented Programming (LOP). A further related area, Grammarware, is discussed in the context of software evolution in Section 3.2.3. DSLs and LOP are closely related to the research central to this thesis. Other areas relating to MDE but less relevant to this thesis, such as formal methods, are not considered here.

### 2.4.1 Domain-Specific Languages

For a set of related problems, a specific, tailored approach is likely to provide better results than instantiating a generic approach for each problem [Deursen *et al.* 2000]. The set of problems for which the specific approach outperforms the generic approach is termed the *domain*. A *domain-specific programming language* (often called a *domain-specific language* or *DSL*) enables the encoding of solutions for a particular domain.

Like modelling languages, DSLs describe abstract syntax. Furthermore, a common language can be used to define DSLs (e.g. EBNF [ISO/IEC 1996]), like the use of MOF for defining modelling languages. In addition to abstract syntax, DSLs typically define a textual concrete syntax but, like modelling languages, can utilise a graphical concrete syntax.

Cobol, Fortran and Lisp first existed as DSLs for solving problems in the domains of business processing, numeric computation and symbolic processing respectively, and evolved to become general-purpose programming languages [Deursen *et al.* 2000]. SQL, on the other hand, is an example of a DSL that, despite undergoing much change, has not grown into a general-purpose language. Unlike a general-purpose language, a single DSL cannot be used to program an entire application. DSLs are often small languages at inception, but can grow to become complicated (such as SQL). Within their domain, DSLs should be easy to read, understand and edit [Fowler 2010].

There are two ways in which DSLs are typically implemented. An *internal* DSL uses constructs from a general-purpose language (the *host*) to describe the domain [Fowler 2010]. Examples of internal DSLs include the libraries of abstract data types that are part of many programming languages (e.g. STL for C++, the Collections API for Java). Some languages are better than others for hosting internal DSLs. For example, Ruby has been proposed as a suitable host for DSLs due to its unintrusive syntax and flexible runtime evaluation [Fowler 2010, ch. 4]. In Lisp, internal DSLs can be implemented by using macros to translate domain-specific concepts to Lisp abstractions [Graham 1993].

When the gap between domain and programming concepts is large, constructing an internal DSL can require a lot of programming effort. Consequently, *translating* DSL programs into code written in a general-purpose language has been recommended [Parr 2007]. The term *external* is sometimes used for this style of DSL implementation [Fowler 2010]. Programs written in simple DSLs are often easy to translate to programs in an existing general-purpose language [Parr 2007]. Approaches to translation include preprocessing; building or generating an interpreter or compiler; or extending an existing compiler or interpreter [Fowler 2010].

The construction of an external DSL can be achieved using many of the principles, practices and tools used in MDE. Parsers can be generated using text-to-model transformation; syntactic constraints can be specified with model validation; and translation can be specified using model-to-model and model-to-text transformation. MDE tools are used to implement two external DSLs in Chapter 5.

Internal and external DSLs have been successfully used as part of application development in many domains [Deursen *et al.* 2000]. They have been used in conjunction with general-purpose languages to build systems rapidly and to improve productivity in the development process (such as automation of system deployment and configuration). More recently, some developers are building complete applications by combining DSLs, in a style of development called Language-Oriented Programming.

### 2.4.2 Language-Oriented Programming

DSLs are central to LOP, a style of software development [Ward 1994]. Firstly, a very high-level language to encode problem domains is developed. Simultaneously, a compiler is developed to translate programs written in the high-level language to an existing programming language. Ward describes how this approach to programming can enhance the productivity of development and the understandability of a system. Additionally, Ward mentions the way in which multiple very high-level languages could be layered to separate domains.

Combining DSLs to solve a problem is not a new technique [Fowler 2010]. Traditionally, UNIX has encouraged developers to combine programs written in small (domain-specific) languages (such as awk, make, sed, lex and yac) to solve problems. Lisp, Smalltalk and Ruby programmers often construct domain-specific languages when developing programs [Graham 1993].

To fully realise the benefits of LOP, the development effort required to construct

DSLs must be minimised. Two approaches for constructing DSLs seem to be prevalent for LOP. The first advocates using a highly dynamic, reflexive and extensible programming language to specify DSLs. This category of language has been termed a *superlanguage* [Clark *et al.* 2008]. The superlanguage permits new DSLs to re-use constructs from existing DSLs, which simplifies development.

A *language workbench* [Fowler 2010, ch. 9] is an alternative means for simplifying DSL development. Language workbenches provide tools, wizards and DSLs for defining abstract and concrete syntax, for constructing editors and for specifying code generators.

For defining DSLs, the main difference between using a language workbench or a superlanguage is the way in which semantics of language concepts are encoded. In a language workbench, a typical approach is to write a generator for each DSL [Fowler 2010], whereas a superlanguage often requires that semantics be encoded in the definition of language constructs [Clark *et al.* 2008].

Like MDE, LOP requires mature and powerful tools and languages to be applicable in the large, and to complex systems. Unlike MDE, LOP tools typically combine concrete and abstract syntax. The emphasis for LOP is in defining a single, textual concrete syntax for a language. MDE tools might provide more than one concrete syntax for a single modelling language. For example, two distinct concrete syntaxes are used for the tree-based and graphical editors of the simple state-machine language shown in Figures 2.7 and 2.11.

Some of the key concerns for MDE are also important to the success of LOP. For example, tools for performing LOP and MDE need to be as usable as those available for traditional development, which often include support for code-completion, automated refactoring and debugging. Presently, these features are often lacking in tools that support LOP or MDE.

In summary, LOP addresses many of the same issues with traditional development as MDE, but requires a different style of tool. LOP focuses more on the integration of distinct DSLs, and providing editors and code generators for them. Compared to LOP, MDE typically provides more separation between concrete and abstract syntax, and concentrates more on model management.

### 2.4.3 Summary

This section has described two areas of research related to MDE, domain-specific languages (DSLs) and language-oriented programming (LOP). DSLs facilitate the encoding of solutions for a particular problem domain. For solving problems in their domain, DSLs can be easier to read, use and edit than general-purpose programming languages [Deursen *et al.* 2000, Fowler 2010]. During MDE, one or more DSLs may be used to model the domain, and the tools and techniques for implementing DSLs can be used for MDE.

LOP is an approach to software development that seeks to specify complete systems using a combination of DSLs. Contemporary LOP seeks to minimise the effort required to specify and use DSLs. Like MDE, LOP requires mature and powerful tools, but, unlike MDE, LOP does not separate concrete and abstract syntax, and does not focus on model management, which is a key development activity in MDE.

## 2.5 Benefits of and Current Challenges to MDE

Compared to traditional software engineering approaches and to domain-specific languages and language-oriented programming, MDE has several benefits and weaknesses. This section identifies benefits of and challenges to MDE, synthesised from the literature reviewed in this chapter.

### 2.5.1 Benefits

Two benefits of MDE are now identified, and used to describe the advantages of the MDE principles and practices discussed in this chapter.

**Tool interoperability** MOF, the standard MDE metamodelling language, facilitates interoperability between tools via model interchange. With Ecore, EMF provides a reference implementation of MOF which provides a foundation for many contemporary MDE tools. Interoperability between modelling tools allows model management to be performed across a range of tools, and developers are not tied to one vendor. Furthermore, models represented in a range of modelling languages can be used together in a single environment. Prior to the formulation of MOF, developers would use different tools for each modelling language. Each tool would probably have different storage formats, complicating the interchange of models between tools.

**System evolution** The guidelines set out for MDE in MDA [OMG 2008b] highlight principles and patterns for modelling to increase the adaptability of software systems by, for example, separating platform-specific and platform-independent detail. When the target platform changes (for example a new technological architecture is required), only part of the system needs to be changed. The platform-independent detail can be re-used wholesale.

Related to this, MDE facilitates automation of the error-prone or tedious elements of software engineering. For example, code generation can be used to automatically produce so-called “boilerplate” code, which is repetitive code that cannot be restructured to remove duplication (typically for technological reasons).

While MDE can be used to reduce the extent to which a system is changed in some circumstances, MDE also introduces additional challenges for managing system evolution [Mens & Demeyer 2007]. For example, mixing generated and hand-written code typically requires a more elaborate software architecture than would be used for a system composed of only hand-written code. Further examples of the challenges that MDE presents for evolution are discussed in the sequel.

### 2.5.2 Challenges

Three challenges for MDE are now identified, and used to motivate areas of potential research for improving MDE. The remainder of the thesis focuses on the final challenge, maintainability in the small.

**Learnability** MDE involves new terminology, development activities and principles for software engineering. For the novice, producing a simple system with MDE is arguably challenging. For example, GMF is difficult for new users to understand and mechanisms for its simplification have been proposed [Kolovos *et al.* 2010]. It seems reasonable to assume that the extent to which MDE tools and principles can be learnt will eventually determine the adoption rate of MDE.

**Scalability** In traditional approaches to software engineering, a model is considered of comparable value to any other documentation artefact, such as a word processor document or a spreadsheet [Rose *et al.* 2010c]. As a result, the convenience of maintaining self-contained model files which can be easily shared outweighs other desirable attributes. This perception is thought to have led to the situation where single-file models of the order of tens (if not hundreds) of megabytes, containing hundreds of thousands of model elements, are the norm for real-world software projects [Kolovos *et al.* 2008a].

MDE languages and tools must scale such that they can be used with large and complex models. Ways in which the scalability of model management tasks, such as model transformation, can be improved have been explored [Hearnden *et al.* 2006, Ráth *et al.* 2008, Tratt 2008]. Kolovos prescribes a different approach, suggesting that MDE research should aim for greater modularity in models, which, as a by-product, will result in greater scalability in MDE [Kolovos *et al.* 2008a]. Scalability of MDE tools is a key concern for practitioners and, for this reason, scalability has been called the "holy grail" of MDE [Kolovos *et al.* 2008a].

**Maintainability in the small** Notwithstanding the benefits of MDE for managing the evolution of systems, the introduction of additional development artefacts (such as models and metamodels) and activities (such as model transformation and validation) presents additional challenges for the way in which software evolution is managed [Mens & Demeyer 2007]. For example, in traditional approaches to software engineering, maintainability is achieved by restructuring code, updating documentation and regression testing [Feathers 2004]. It is not yet clear the extent to which existing maintenance activities can be applied in MDE. (For example, should models be tested and, if so, how?)

As demonstrated in Chapter 4, the way in which some MDE tools are structured limits the extent to which some traditional maintenance activities can be performed. Understanding, improving and assessing the way in which evolution is managed in the context of MDE is an open research topic to which this thesis contributes.

### 2.5.3 Summary

This section has identified some of the benefits of and challenges for contemporary MDE. The interoperability of tools and modelling languages in MDE allows developers greater flexibility in their choice of tools and facilitates interchange between heterogeneous tools and modelling frameworks. MDE is more flexible than other, more formal approaches to software engineering, which can be beneficial for constructing complex systems. The principles and practices of MDE can be used to achieve greater maintainability of systems by, for example, separating platform-independent and platform-specific details.

As MDE tools approach maturity, non-functional requirements, such as learnability, and scalability, become increasingly desirable for practitioners. MDE tools must also be able to support developers in managing changing software. This section has demonstrated some of the weakness of contemporary MDE, particularly in the areas of learnability, scalability and maintainability.

## 2.6 Chapter Summary

This chapter has discussed Model-Driven Engineering (MDE), a state-of-the-art and principled approach to software engineering. The terminology, development activities and tools used in a typical MDE process were introduced. Two areas relating to MDE, language-oriented programming and domain-specific languages, were discussed, and three methods for performing MDE were reviewed.

Traditional approaches to software engineering do not treat modelling artefacts – such as model, metamodels and model management operations – as first-class citizens, and they are typically represented in an unstructured manner, if at all. MDE involves creating, manipulating and managing changes to modelling artefacts and therefore modelling artefacts are represented in a structured manner. This chapter has demonstrated that contemporary MDE tools, such as EMF and Epsilon, provide structures and processes for creating and manipulating modelling artefacts, but not for managing evolutionary change. Chapters 3 and 4 review, explore and investigate structures and processes for managing the evolution of modelling artefacts.

# Chapter 3

# Review of Software Evolution

This chapter reviews existing work on software evolution. In particular, this chapter explores the ways in which software evolution is identified and managed in the context of MDE, which was discussed in Chapter 2. Potential directions for future research are identified from the literature review, and Chapter 4 performs a critical analysis of some of the tools and techniques reviewed in this chapter. The principles of software evolution are discussed in Section 3.1, while Section 3.2 reviews the ways in which evolution is identified, analysed and managed in a range of fields, including relational databases, programming languages, and MDE development environments. From the review, Section 3.4 synthesises research challenges for software evolution in the context of MDE, highlighting those to which this thesis contributes, and elaborates on the research method used in this thesis.

## 3.1 Software Evolution Theory

Software evolution is an important facet of software engineering. As discussed in Section 1.2, studies [Erlikh 2000, Moad 1990] have shown that the evolution of software can account for as much as 90% of a development budget, and there is no reason to believe the situation is different today. Such figures are sometimes described as uncertain [Sommerville 2006, ch. 21], primarily because the term evolution is not used consistently. For example, some authors prefer the term *maintenance* to *evolution*. Here, the latter is preferred as the former can imply deterioration caused by repeated use, and most software systems do not deteriorate in this manner [Ramil & Lehman 2000]. There is a corpus of software evolution research. Publications in this area have existed since the 1960s (e.g. [Lehman 1969]).

The remainder of this section introduces software evolution terminology and discusses three research areas that relate to software evolution: refactoring, design patterns and traceability. Refactoring concentrates on improving the structure of existing systems, design patterns on best practices for software design, and traceability for recording and analysing the lifecycle of software artefacts. Each area provides a vocabulary for discussing software design and evolution. There is an abundance of research in these areas, including seminal works on refactoring, such as [Opdyke 1992] and [Fowler 1999]; and on design patterns, such as [Gamma *et al.* 1995].

### 3.1.1 Categories of Software Evolution

Three activities – addressing changing requirements, adapting to new technologies, and architectural restructuring – have been identified as the primary causes of software evolution [Sjøberg 1993]. These activities are the motivations for three common types of software evolution [Sommerville 2006, ch. 21]:

- **Corrective evolution** takes place when a system exhibiting unintended or faulty behaviour is corrected. Alternatively, corrective evolution may be used to adapt a system to new or changing requirements.
- **Adaptive evolution** is employed to make a system compatible with a change to platforms or technologies that underpin its implementation.
- **Perfective evolution** refers to the process of improving the internal quality of a system, while preserving the behaviour of the system.

The remainder of this section adopts this categorisation for discussing software evolution literature. Refactoring (discussed in Section 3.1.3), for instance, is one way in which perfective evolution can be realised.

### 3.1.2 Evolutionary Development Activities

Software evolution is identified and managed via many different development activities. Two software evolution activities, *impact analysis* (for reasoning about the effects of evolution) and *change propagation* (for updating one artefact in response to a change made to another) are commonplace [Winkler & Pilgrim 2010]. In addition, *reverse engineering* (analysing existing development artefacts to extract information) and *source code translation* (rewriting code to use a more suitable technology, such as a different programming language) are also important software evolution activities [Sommerville 2006].

MDE introduces new software engineering principles, and the concerns and requirements for software evolution differ between MDE processes and traditional software development processes. For example, MDE facilitates portable software by prescribing platform-independent and platform-specific models (as discussed in Section 2.1.4), and as such source code translation is arguably less relevant to MDE than to traditional software engineering. Because MDE seeks to capture the essence of the software in models, reverse engineering information from, for example, code is also less likely to be relevant to MDE than to traditional software engineering. Consequently, this thesis focuses on impact analysis and change propagation, which are both relevant to MDE [Winkler & Pilgrim 2010], as discussed in Section 3.2.4.

### 3.1.3 Refactoring

There is "an urgent need for techniques that reduce software complexity by incrementally improving the internal software quality" [Mens & Tourwé 2004]. Refactoring is "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fowler 1999, pg. xvi]. Refactoring plays a significant role in the evolution of software systems – a recent study of five open-source projects showed that over 80% of changes were refactorings [Dig & Johnson 2006b].

Typically, refactoring literature concentrates on three primary activities in the refactoring process: *identification* (where should refactoring be applied, and which refactorings should be used?), *verification* (has refactoring preserved behaviour?) and *assessment* (how has refactoring affected other qualities of the system, such as cohesion and efficiency?).

Beck describes an informal means for identifying the need for refactoring, termed *bad smells*: "structures in the code that suggest (sometimes scream for) the possibility of refactoring" [Fowler 1999, foreword]. Tools and semi-automated approaches have also been devised for refactoring identification, such as Daikon [Kataoka *et al.* 2001], which detects program invariants that may indicate the possibility for refactoring. Clone analysis tools have been employed for identifying refactorings that eliminate duplication [Balazinska *et al.* 2000, Ducasse *et al.* 1999]. The types of refactoring being performed may vary over different domains. For example, Buck[1] describes refactorings, such as "Skinny Controller, Fat Model", particular to the Ruby on Rails web framework[2].

Since 2006, Dig has been studying the refactoring of systems that are developed by combining components, possibly developed by different organisations. The changes made to five components that are known to have been re-used often were categorised, with the hypothesis that a significant number of the changes could be classified as behaviour-preserving (i.e. refactorings) [Dig & Johnson 2006b]. By using examples from the survey, an algorithm was devised for automatically detecting refactorings to a high degree of accuracy (over 85%) [Dig *et al.* 2006]. The algorithm was then utilised in tools for (1) replaying refactorings to perform migration of client code following breaking changes to a component [Dig & Johnson 2006a], and (2) versioning object-oriented programs using a refactoring-aware configuration management system [Dig *et al.* 2007]. The latter facilitated better understanding of program evolution, and the refinement of the refactoring detection algorithm.

### 3.1.4 Patterns and Anti-Patterns

A *design pattern* identifies a commonly occurring design problem and describes a reusable solution to that problem. Related design patterns are combined to form a *pattern catalogue* – such as for object-oriented programming [Gamma *et al.* 1995] or enterprise

---

[1] In a keynote address to the First International Ruby on Rails Conference (RailsConf), May 2007, Portland, Oregon, United States of America.

[2] `http://www.rubyonrails.org`

applications [Fowler 2002]. A pattern description comprises at least a name, overview of the problem, and details of a common solution [Brown *et al.* 1998]. Depending on the domain, further information may be included in the pattern description (such as a classification, a description of the pattern's applicability and an example usage).

Design patterns can be thought of as describing objectives for improving the internal quality of a system (i.e. perfective software evolution). Often developers improve the quality of systems by restructing systems to exhibit design patterns [Kerievsky 2004]. Studying the way in which experts perform perfective software evolution can lead to devising best practices, sometimes in the form of a pattern catalogue, such as the object-oriented refactorings described in [Fowler 1999].

Design patterns were first used to devise a pattern catalogue for town planning [Alexander *et al.* 1977]. Refactorings were later adapted to software architecture, by specifying a pattern catalogue for designing user-interfaces [Beck & Cunningham 1989]. Utilising pattern catalogues allowed the software industry to "reuse the expertise of experienced developers to repeatedly train the less experienced" [Brown *et al.* 1998, pg. 10]. Furthermore, design patterns "help to define a vocabulary for talking about software development and integration challenges; and provide a process for the orderly resolution of these challenges" [Rising 2001, pg. xii].

Anti-patterns are an alternative literary form for describing patterns of a software architecture [Brown *et al.* 1998]. Rather than describe patterns that have often been observed in successful architectures, they describe those which are present in unsuccessful architectures. Essentially, an anti-pattern is a pattern in an inappropriate context, which describes a problematic solution to a frequently encountered problem. The (anti-)pattern catalogue may include alternative solutions that are known to yield better results (termed "refactored solutions" by Brown). Catalogues might also consider the reasons why (inexperienced) developers use an anti-pattern. Brown notes that "patterns and anti-patterns are complementary" [Brown *et al.* 1998, pg. 13]; both are useful in providing a common vocabulary for discussion of system architectures and in educating less experienced developers.

### 3.1.5 Traceability

A software development artefact rarely evolves in isolation. Changes to one artefact cause and are caused by changes to other artefacts (e.g. object code is recompiled when source code changes, source code and documentation are updated when requirements change). Hence, traceability – the ability to describe and follow the life of software artefacts [Winkler & Pilgrim 2010, Lago *et al.* 2009] – is closely related to and facilitates software evolution.

Historically, traceability is a branch of requirements engineering, but is increasingly used for artefacts other than requirements [Winkler & Pilgrim 2010]. Because MDE prescribes automated transformation between models, traceability is also researched in the context of MDE. The remainder of this section discusses traceability principles focussing on the relationship between traceability and software evolution. Section 3.2.4 reviews the traceability literature that relates to MDE.

The traceability literature uses inconsistent terminology. This thesis uses the terminology of a survey of traceability research [Winkler & Pilgrim 2010]: *traceability* is the ability to describe and follow the life of software artefacts; *traceability links* are the relationships between software artefacts. Traceability is facilitated by traceability links, which document the dependencies, causalities and influences between artefacts. Traceability links are established by hand or by automated analysis of artefacts. In MDE environments, some traceability links can be automatically inferred because the relationships between some types of artefact are specified in a structured manner (for example, as a model-to-model transformation).

Traceability links are defined between artefacts at the same level of abstraction (horizontal links) and at different levels of abstraction (vertical links). Traceability links can be uni- or bi-directional. The former are navigated either *forwards* (away from the dependent artefact) or *backwards* (towards the dependent artefact). Figure 3.1 summaries these categories of traceability link.

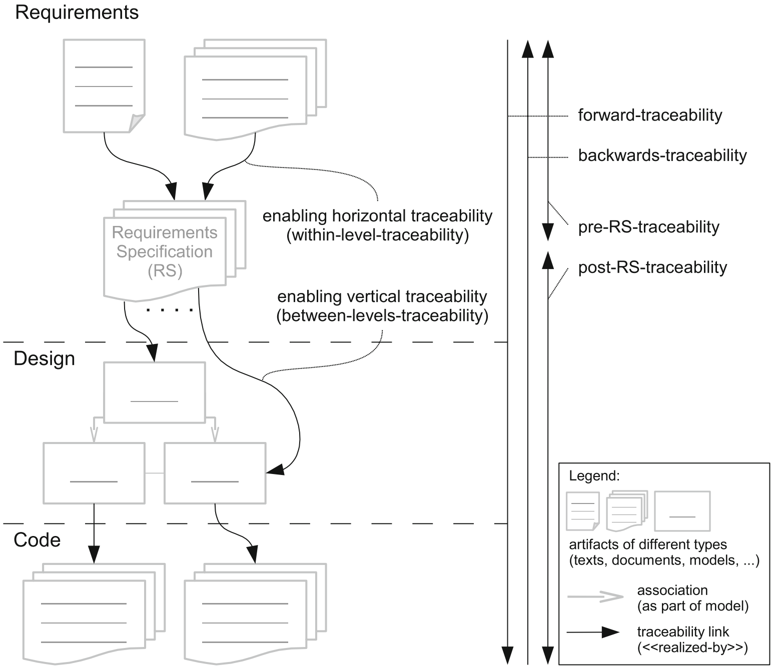


Figure 3.1: Categories of traceability link [Winkler & Pilgrim 2010].

Traceability supports software evolution activities, such as impact analysis (discovering and reasoning about the effects of a change) and change propagation (updating impacted artefacts following a change to an artefact). Moreover, automated software evolution is facilitated by programmatic access to traceability links.

Current approaches for traceability-supported software evolution use *triggers* and *events*. Each approach proposes mechanisms for detecting triggers (changes to artefacts) and for notifying dependent artefacts of events (the details of a change). Existing approaches vary in the extent to which they automatically update dependent artefacts. Some approaches [Chen & Chou 1999, Cleland-Huang *et al.* 2003] do not perform automatic updates, while others [Aizenbud-Reshef *et al.* 2005, Costa & Silva 2007] provide guided or fully automatic updates. Section 3.2.4 provides a more thorough discussion and critical analysis of event-based approaches for impact analysis and change propagation in the context of MDE.

To remain accurate and hence useful, traceability links must be updated as a system evolves. Although most existing approaches to traceability are "not well suited to the evolution of [traceability] artefacts" [Winkler & Pilgrim 2010, pg. 24], there is some work in this area. For example, Mader et al. describe a development environment that records changes to artefacts, comparing the changes to a catalogue of built-in patterns [Mäder *et al.* 2008]. Each pattern provides an executable specification for updating traceability links.

Software evolution and traceability are entangled concerns. Traceability facilitates software evolution activities such as impact analysis and change propagation. Traceability is made possible with consistent and accurate traceability links. Software evolution can affect the relationships between artefacts (i.e. the traceability links) and hence software evolution techniques are applied to ensure that traceability links remain consistent and accurate.

## 3.2 Software Evolution in Practice

Using the principles of software evolution described above, this section examines the ways in which evolution is identified, managed and analysed in a variety of settings, including programming languages, grammarware, relational database management systems and MDE.

### 3.2.1 Programming Language Evolution

Programming language designers often attempt to ensure that legacy programs continue to conform to new language specifications. For example, the designers of the Java [Gosling *et al.* 2005] language are reluctant to introduce new keywords (becauae identifiers in existing programs could then be mistakenly recognised as instances of the new keyword) [Cervelle *et al.* 2006].

Although designers are cautious about changing programming languages, evolution does occur. In this section, two examples of the ways in which programming languages have evolved are discussed. The vocabulary used to describe the scenarios is applicable to evolution of MDE artefacts. Furthermore, MDE sometimes involves the use of general-purpose modelling languages, such as UML [OMG 2007a]. The evolution of general-purpose modelling languages may be similar to that of general-purpose programming languages.

### Reduction

Mapping language abstractions to executable concepts can be complicated. Therefore, languages are sometimes evolved to simplify the implementation of translators (compilers, interpreters, etc). It seems that this type of evolution is more likely to occur when language design is a linear process (with a reference implementation occurring after design), and in larger languages.

The evolution of FORTRAN involved some simplification of language constructs [Backus 1978]. Originally, FORTRAN's `DO` statements were awkward to compile. The semantics of `DO` were simplified such that more efficient object code could be generated from them. Essentially, the simplified `DO` statement allowed linear changes to index statements to be detected (and optimised) by compilers.

The removal of the `RELABEL` construct (which simplified indexing into multidimensional arrays) from FORTRAN [Backus 1978] is a further example of reduction.

### Revolution

When using a programming language, software engineers often develop best practices, which are expressed and shared with other software engineers. Design patterns are one way in which best practices may be communicated to other developers. Incorporating existing design patterns as language constructs is one approach to specifying a new language (e.g. [Bosch 1998]).

Lisp makes idiomatic some of the Fortran List Processing Language (FLPL) design patterns. For example, the awkwardness of using FLPL's `IF` construct led experienced developers to define the function `XIF(P, T, F)` where `T` was executed iff `P` was true, and `F` was executed otherwise [McCarthy 1978]. However, such functions had to be used sparingly, as all three arguments would be evaluated due to the way in which FORTRAN executed function calls. A more efficient semantics, wherein `T` (`F`) was only evaluated when `P` was true (false), was inspired by the use of `XIF` [McCarthy 1978]. Because FORTRAN programs could not express these semantics, McCarthy's new construct informed the design of Lisp. Lazy evaluation in functional languages can be regarded as a further step on this evolutionary path.

## 3.2.2 Schema Evolution

This section reviews schema evolution research. Work covering the evolution of XML and database schemata is considered. Both types of schema are used to describe a set of concepts (termed the *universe of discourse* in database literature). Schema designers decide which details of their domain concepts to describe; their schemata provide an abstraction containing only those concepts which are relevant [Elmasri & Navathe 2006, pg. 30]. As such, schemata in these domains may be thought of as analogous to metamodels – they provide a means for describing an abstraction over a phenomenon of interest. Therefore, approaches to identifying, analysing and performing schema evolution are directly relevant to the evolution of metamodels in MDE. However, the patterns of evolution commonly seen in database systems and with XML may be different to those of metamodels because evolution can be:

- **Domain-specific**: Patterns of evolution may be applicable only within a particular domain (e.g. normalisation in a relational database).
- **Language-specific**: The way in which evolution occurs may be influenced by the language (or tool) used to express the change. (For example, some implementations of SQL may not have a `rename relation` command, so alternative means for renaming a relation must be used).

Many of the published works on schema evolution share a similar method, with the aim of defining a taxonomy of evolutionary operators. Developers are expected to apply the operators to evolve schemata. This approach is used heavily for XML schema evolution (e.g. [Guerrini *et al.* 2005, Kramer 2001, Su *et al.* 2001]). Similar taxonomies exist for schema evolution in relational database systems (e.g. in [Banerjee *et al.* 1987, Edelweiss & Moreira 2005]), but other approaches to evolution are also prevalent. An alternative is discussed in depth, along with a summary of other work.

### XML Schema Evolution

XML provides a specification for defining mark-up languages. XML documents can reference a schema, which provides a description of the ways in which the concepts in the mark-up should relate (i.e. the schema describes the syntax of the XML document). Prior to the definition of the XML Schema specification [W3C 2007] by the World Wide Web Consortium (W3C)[3], authors of XML documents could use a specific Document Type Definition (DTD) to describe the syntax of their mark-up language. XML Schemata provide several advantages over the DTD specification, including the following:

- XML Schemata are defined in XML and may, therefore, be validated against another XML Schema. DTDs are specified in another language entirely, which requires a different parser and different validation tools.
- DTDs provide a means for specifying constraints only on the mark-up language, whereas XML Schemata may also specify constraints on the data in an XML document.

Work on the evolution of the structure of XML documents is now discussed. Existing work concentrates on changes made to XML Schema (e.g. [Guerrini *et al.* 2005]) or to DTDs (e.g. [Kramer 2001]).

Guerrini et al. propose a set of primitive operators for changing XML schemata [Guerrini *et al.* 2005]. The set is sound (application of an operator always results in a

---

[3] `http://www.w3.org/`

valid schema) and complete (any valid schema can be produced by composing operators). Guerrini et al. also detail those operators that can be 'validity-preserving' (i.e. application of the operator produces a schema that does not require its instances to be migrated). The arguments of an operator influence whether it is validity-preserving. For example, inserting an element is validity-preserving when the lowerbound of the new element is zero, but not otherwise. In addition to soundness and completeness, minimality is another desirable property in a taxonomy of primitive operators for performing schema evolution [Su *et al.* 2001]. To complement a minimal set of primitives, and to improve the conciseness with which schema evolutions can be specified, Guerrini et al. propose a number of 'high-level' operators, which comprise two or more primitive operators.

Kramer provides a further taxonomy of primitives for the evolution of XML schema [Kramer 2001]. To describe her evolution operators, Kramer uses a template, which comprises a name, syntax, semantics, preconditions, resulting DTD changes and resulting data changes section for each operator. This style is similar to a pattern catalogue, but Kramer does not provide a context for her operators (i.e. there are no examples that describe when the application of an operator may be useful). Kramer utilises her taxonomy in a repository system, Exemplar, for managing the evolution of XML documents and their schemata. The repository provides an environment in which the variation of XML documents can be managed. However, to be of practical use, Exemplar would benefit from integration with a source code management system (to provide features such as branching, and version merging).

The evolutionary taxonomies described in the above approaches are complete in the sense that any valid schema can be produced, but do not allow for arbitrary updates of the XML documents in response to schema changes [Pizka & Jürgens 2007]. Hence, none of the approaches discussed in this section ensure that information contained in XML documents is not lost.

### Relational Database Schema Evolution

Defining a taxonomy of operators for performing schema updates is also common for supporting relational database schema evolution. For example, two taxonomies are described in [Edelweiss & Moreira 2005, Banerjee *et al.* 1987]. However, problems that arise when performing data migration after these taxonomies have been used to specify schema evolution have been highlighted:

> "There are two major issues involved in schema evolution. The first issue is understanding how a schema has changed. The second issue involves deciding when and how to modify the database to address such concerns as efficiency, availability, and impact on existing code. Most research efforts have been aimed at this second issue and assume a small set of schema changes that are easy to support, such as adding and removing record fields, while requiring the maintainer to provide translation routines for more complicated changes. As a result, progress has been made in developing the

> backend mechanisms to convert, screen, or version the existing data, but little progress has been made on supporting a rich collection of changes" [Lerner 2000, pg. 84].

Fundamentally , Lerner believes that any taxonomy of operators for schema evolution is too fine-grained to capture the semantics intended by the schema developer, and therefore cannot be used to provide automated migration: existing taxonomies are concerned with the "editing process rather than the editing result" [Lerner 2000]. Furthermore, Lerner believes that developing such a taxonomy creates a proliferation of operators, increasing the complexity of specifying migration. To demonstrate, Lerner considers moving a field from one type to another in a schema. This could be expressed using two primitive operators, `delete_field` and `add_field`. However, the semantics of a `delete_field` command likely dictate that the data associated with the field will be lost, making it unsuitable for use when specifying that a type has been moved. The designer of the taxonomy could introduce a `move_field` command to solve this problem, but now the maintainer of the schema needs to understand the difference between the two ways in which moving a type can be specified, and carefully select the correct one. Lerner provides other examples which expand on this issue (such as introducing a new type by splitting an existing type). Even though Lerner highlights that a fine-grained approach may not be the most suitable for specifying schema evolution, other potential uses for a taxonomy of evolutionary operators (such as, a common vocabulary for discussing the restructuring of a schema) are not discussed.

Lerner proposes an alternative to operator-based schema evolution in which two versions of a schema are compared to infer the schema changes. Using the inferred changes, migration strategies for the affected data can be proposed. Lerner presents algorithms for inferring changes from schemata and performing both automated and guided migration of affected data. By inferring changes, developers maintaining the schema are afforded more flexibility. In particular, they need not use a domain-specific language or editor to change a schema, and can concentrate on the desired result, rather than how best to express the changes to the schema in the small. Furthermore, algorithms for inferring changes have uses other than for migration (e.g. for semantically-aware comparison of schemata, similar to that provided by a refactoring-aware *source code management system* [Dig *et al.* 2007]). Comparison of two schema versions might suggest more than one feasible strategy for updating data, and Lerner does not propose a mechanism for distinguishing between feasible alternatives.

Vries et al. propose the introduction of an extra layer to the typical architecture of a relational database management system [Vries & Roddick 2004]. They demonstrate the way in which the extra layer can be used to perform migration subsequent to a change of an attribute type. The layer contains (mathematical) relations, termed *mesodata*, that describe the way in which an old value (data prior to migration) maps to one or more new values (data subsequent to migration). These mappings are added to the mesodata by the developer performing schema updates, and are used to semi-automate migration. It is not clear how this approach can be applied when schema evolution is not an attribute type change.

In the O2 database [Ferrandina *et al.* 1995], schema updates are performed using a domain-specific language. Modification constructs are used to describe the changes to be made to the schema. To perform data migration, O2 provides conversion functions as part of its modification constructs. Conversion functions are either user-defined or default (pre-defined). The pre-defined functions concentrate on providing mappings for attributes whose types are changed (e.g. from a double to an integer; from a set to a list). Additionally, conversion functions may be executed in conjunction with the schema update, or they may be deferred, and executed only when the data is accessed through the updated schema. Ferrandina et al. observe that deferred updates may prevent unnecessary downtime of the database system. Although the approach used in the O2 database addresses the concern that "approaches to coping with schema evolution should be concerned with the editing result rather than the editing process" [Lerner 2000], there is no support for some types of evolution such as moving an attribute from one relation to another.

### 3.2.3 Grammar Evolution

An engineering approach to producing grammarware (grammars and software that depends on grammars, such as parsers and program convertors) is regarded as one way to better support software development [Klint *et al.* 2003]. The grammarware engineering approach envisaged by Klint et al. is based on best practices and techniques, which they anticipate will be derived from addressing open research challenges. Klint et al. identify seven key questions for grammarware engineering, one of which relates to grammar evolution: "How does one systematically transform grammatical structure when faced with evolution?" [Klint *et al.* 2003, pg. 334].

Between 2001 and 2005, Ralf Lämmel (an author of [Klint *et al.* 2003]) and his colleagues at Vrije Universiteit published several important papers on grammar evolution. Lämmel has proposed a taxonomy of operators for semi-automatic grammar refactoring [Lämmel 2001] and demonstrated their usefulness in recovering the formal specifications of undocumented grammars (such as VS COBOL II in [Lämmel & Verhoef 2001]) and in specifying generic refactorings [Lämmel 2002].

The work of Lämmel et al. focuses on grammar evolution for refactoring or for *grammar recovery* (corrective evolution in which a deviation from a language reference is removed), but does not address the impact of grammar evolution on corresponding programs or grammarware. For instance, when a grammar changes, updates are potentially required both to programs written in that grammar and to tools that parse, generate or otherwise manipulate programs written in that grammar.

Grammar-program co-evolution has been recognised is a challenge for grammarware [Pizka & Jürgens 2007]. Pizka and Juergens believe that most grammars evolve over time and that, without tool support, co-evolution is a complex, time-consuming and error prone task. To this end, the language evolution tool, Lever [Pizka & Jürgens 2007], defines and uses operators for changing grammars (and programs).

Lever can be used to manage the evolution of grammars and programs, and also the co-evolution of grammars and programs. By contrast Lämmel's taxonomy

[Lämmel 2001] can be used only to manage grammar evolution. However, as a consequence, Lever sacrifices the formal preservation properties of Lämmel's taxonomy.

### 3.2.4 Evolution of MDE Artefacts

As discussed in Chapter 1, the evolution of development artefacts during MDE inhibits the productivity and maintainability of model-driven approaches for constructing software systems. Mitigating the effects of evolution on MDE is an open research topic, to which this thesis contributes. This section surveys literature that explores the evolution of development artefacts used when performing MDE, and Chapter 4 contributes a more detailed and critical analysis of the tools and techniques described in this section.

Evolution in MDE is complicated, because MDE involves combining several different types of interdependent artefact [Deursen *et al.* 2007]. More specifically, there are three types of development artefact specific to MDE (models, metamodels, and specifications of model management operations), and a change to one type of artefact can affect other artefacts (possibly of a different type).

Evolution can regarded as either *syntactic* or *semantic* [Sprinkle & Karsai 2004]. In the former, no information is known about the intention of the the evolutionary change. In the latter, a lack of detailed information about the semantics of evolution can reduce the extent to which change propagation can be automated. For example, consider the case where a class is deleted from a metamodel. The following questions typically need to be answered to facilitate evolution:

- Should subtypes of the deleted class also be removed? If not, should their inheritance hierarchy be changed? What is the correct type for references that used to have the type of the deleted class?

- Suppose that the evolving metamodel is the target of a previous model-to-model transformation. Should the data that was previously transformed to instances of the deleted class now be transformed to instances of another metamodel class?

- What should happen to instances of the deleted metamodel class? Perhaps they should be removed too, or perhaps their data should be migrated to new instances of another class.

Tools that recognise only syntactic evolution tend to lack the information required for full automation of evolution activities. Furthermore, tools that focus only on syntax cannot be applied in the face of additive changes [Gruschko *et al.* 2007]. There are complexities involved in recording the semantics of software evolution. For example, the semantics of an impacted artefact need not always be preserved: this is often the case in corrective evolution.

Notwithstanding the challenges described above, MDE has great potential for managing software evolution and automating software evolution activities, particularly because of model transformations (Section 2.1.4). Approaches for managing evolution in other fields, described above, must consider the way in which artefacts are updated when changes are propagated from one artefact to another. Model transformation languages already fulfil this role in MDE. In addition, model transformations provide a (limited) form of traceability between MDE artefacts, which can be used in impact analysis.

This section focuses on the three types of evolution most commonly discussed in MDE literature. *Model refactoring* is used to improve the quality of a model without changing its functional behaviour. *Model synchronisation* involves updating a model in response to a change made in another model, usually by executing a model-to-model transformation. *Model-metamodel co-evolution* involves updating a model in response to a change made to a metamodel. This section concludes by reviewing existing techniques for visualising model-to-model transformation and assessing their usefulness for understanding evolution in the context of MDE.

### Model Refactoring

Refactoring (Section 3.1.3) is a perfective software evolution activity in which the structure and representation of a system is improved without changing its functional behaviour. In a particular domain, some refactoring activities might occur often and, to increase productivity, could benefit from automation. In a MDE process, reoccurring patterns of evolution might be expressed as refactoring transformations on models, and used to document and perhaps automate model refactoring activities.

In model transformation terminology (discussed in Section 2.1.4), a refactoring is an *endogenous*, *in-place* transformation. Refactorings are applied to an artefact (e.g. model, code) producing a semantically equivalent artefact, and hence an artefact that conforms to the same rules and structures as the original. Because refactorings are used to improve the structure of an existing artefact, the refactored artefact typically replaces the original. Endogenous, in-place transformation languages, suitable for refactoring have been implemented with a declarative style based on graph theory [Biermann *et al.* 2006, Porres 2003] and with a hybrid style that mixes declarative and imperative constructs [Kolovos *et al.* 2007].

There are similarities between the structures defined in the MOF metamodelling language and in object-oriented programming languages. For the latter, refactoring pattern catalogues exist (such as [Fowler 1999]), which might usefully be applied to modelling languages. Moha et al. provide a notation for specifying refactorings for MOF and UML models and Java programs in a generic (metamodel-independent) manner [Moha *et al.* 2009]. Because MOF, UML and the Java language share some concepts with the same semantics (such as classes and attributes), Moha et al. show that refactorings can be shared among them, but only consider 3 of the 72 object-oriented refactorings identified by Fowler. To more thoroughly understand metamodel-independent refactoring, a larger number of refactorings and languages should be explored.

Eclipse, an extensible development environment, provides a library for building development tools for textual languages, LTK (language toolkit)[4]. LTK allows developers to specify – in Java – refactorings for their language, which can be invoked

---

[4] `http://www.eclipse.org/articles/Article-LTK/ltk.html`

via the language editor. LTK makes no assumptions on the way in which languages will be structured, and as such refactorings specified with LTK must interact with the modelling framework directly.

The Epsilon Wizard Language (EWL) [Kolovos *et al.* 2007] is a model transformation language tailored for the specification of model refactorings. EWL is part of the Epsilon platform and re-uses the Epsilon Object Language (EOL), which can query, update and navigate models represented in a diverse range of modelling technologies (Section 2.3.2). Consequently, EWL, unlike LTK, abstracts over modelling frameworks.

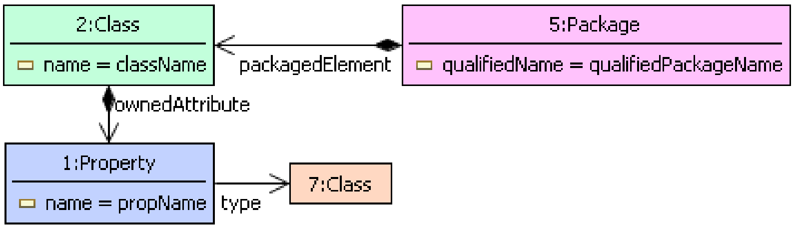
EMF Refactor [Arendt *et al.* 2009] has been compared with EWL and the LTK by specifying a refactoring on a UML model. EMF Refactor, like EWL, contributes a model transformation language tailored for refactoring. Unlike EWL, EMF Refactor has a visual (rather than textual) syntax, and is based on graph transformation concepts. Figure 3.2 shows the "Change attribute to association end" refactoring for the UML metamodel in EMF Refactor. The left-hand side of the refactoring rule, Figure 3.2(a), matches a Class whose owned attributes contains a Property whose type has the same name as a Class. The right-hand side of the rule, Figure 3.2(b), introduces a new Association, whose member end is the Property matched in the left-hand side of the rule. Due to its visual syntax, EMF Refactor might be usable only with modelling technologies based on MOF (which has a graphical concrete-syntax based on UML class diagrams). It is not clear to what extent EMF Refactor can be used with modelling technologies other than EMF.

Existing model refactoring approaches focus on refactoring a model in isolation, rather than *inter-model refactorings*, which affect more than one model at once. The Eclipse Java Development Tools support refactorings of Java code that update many source-code artefacts at once: for example, renaming a class in one source file updates references to that class in other source files. In the context of MDE, support for inter-model refactoring would facilitate a greater degree of model modularisation and hence increase scalability, one of the key challenges for MDE [Kolovos *et al.* 2008a] (Section 2.5.2).

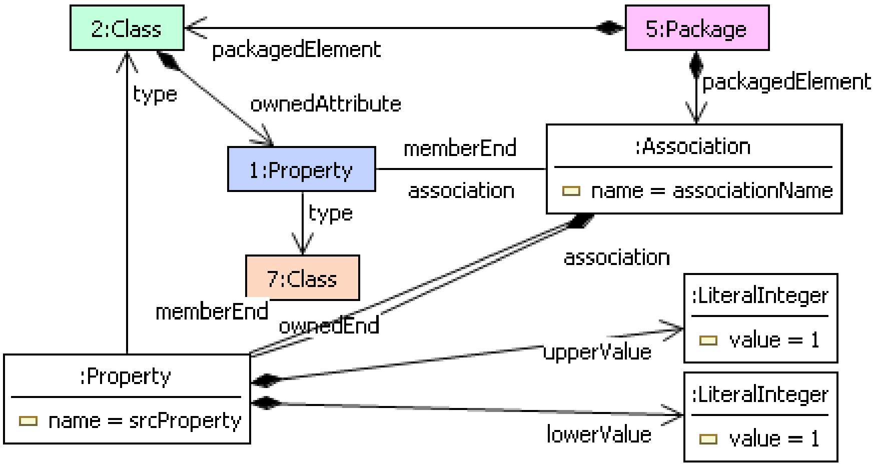
Model refactoring research remains "in its infancy" [Mens *et al.* 2007]. Mens et al. identify formalisms for investigating the feasibility and scalability of model refactoring. In particular, Mens et al. suggest that meaning-preservation (an objective of refactoring, as discussed in Section 3.1.3) can be checked by evaluating OCL constraints, behavioural models or downstream program code.

### Model Synchronisation

Changes made to development artefacts may require the *synchronisation* of related artefacts (models, code, documentation). Traceability links (which capture the relationships of software artefacts) facilitate synchronisation. This section discusses the way in which change propagation is approached in the literature, which typically involves using an incremental style of transformation. Work that addresses more fundamental aspects of model synchronisation, such as capturing trace links and performing impact analysis are also discussed. Finally, synchronisation between models and text



(a) Left-hand side matching rule.



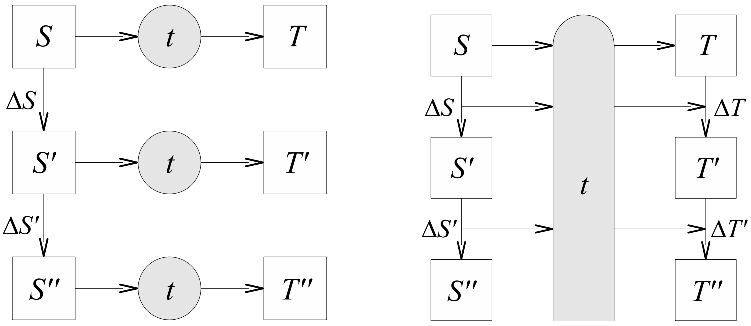
(b) Right-hand side production rule.

Figure 3.2: Attribute to association end refactoring in EMF Refactor. Taken from [Arendt *et al.* 2009].

and between models and trace links is also considered.

**Incremental transformation** Many model synchronisation approaches extend or instrument existing model-to-model transformation languages. Declarative transformation languages are well-suited to the specification of bi-directional transformations and *incremental transformations*, a style of model transformation that facilitates incremental updates of the target model. In fact, much of the model synchronisation literature focuses on incremental transformation.

Incremental transformation can be achieved in one of two ways. Because model-to-model transformation is used to generate one or more target models from one or more source models, when a source model changes, the model-to-model transformation can be invoked to completely re-generate the target models. This activity has been termed *re-transformation* and an alternative, *live transformation*, in which the transformation context is persistent has been proposed [Hearnden *et al.* 2006]. Figure 3.3 illustrates the differences between re-transformation and live transformation, showing the evolution of source and target models on the left-hand and right-hand sides, respectively, and the transformation context in the middle. Live transformation facilitates change propagation from the source to the target models without completely re-generating the target models and is therefore a more efficient approach when changes to the source models affect only part of the target models. Live transformation appears to be a common way to achieve incremental transformation (e.g. [Ráth *et al.* 2008, Tratt 2008]).



(a) Re-transformation. (b) Live transformation.

Figure 3.3: Approaches to incremental transformation. Taken from [Hearnden *et al.* 2006].

Primarily, incremental transformation has been used to address the scalability of model transformations. For large models, transformation execution time can be significantly reduced by using incremental transformation [Hearnden *et al.* 2006]. However, it has been suggested that scalability should be addressed not only by attempting to develop techniques for increasing the speed of model transformation, but also by providing principles, practices and tools for building models that are less monolithic and more modular [Kolovos *et al.* 2008a]. For this end, model synchronisation research should focus on maintainability as well as scalability.

Model synchronisation and incremental transformation can be applied to decouple models and facilitate modularisation [Fritzsche *et al.* 2008], although this is not commonly discussed in the literature. Fritzche et al. contribute a transformation that produces, from any UML model, a model for which performance characteristics can be readily analysed. The relationships between UML and performance model artefacts are recorded using traceability links. The results of the performance analysis are later fed back to the UML model using an incremental transformation made possible by the traceability links. Using this approach, performance engineers can focus primarily on the performance models, while other engineers are shielded from the low-level detail of the performance analysis. As such, Fritzsche et al. show that two different modelling concerns can be separated and decoupled, yet remain cohesive via the application of model synchronisation.

**Towards automated model synchronisation** Some existing work provides a foundation for automating model synchronisation activities. Theoretical aspects of the traceability literature were reviewed in Section 3.1.5, and explored the automated activities that traceability facilitates, such as impact analysis and change propagation. This section now analyses the traceability research in the context of MDE and focuses on the way in which traceability facilitates the automation of model synchronisation activities.

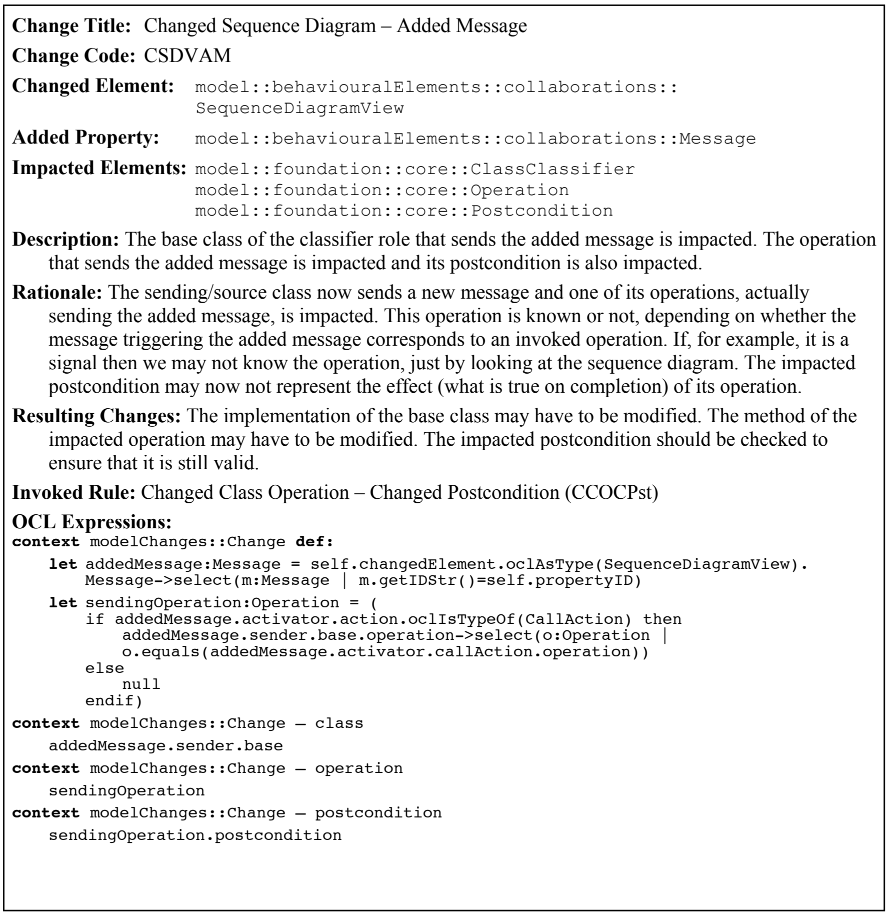
Aside from live transformation, other techniques for capturing trace links between models have been reported. To this end, a model-to-model transformation has been enriched with traceability information using a generic higher-order transformation [Jouault 2005]. Given a transformation, the generic higher-order transformation adds transformation rules that produce a traceability model. In contrast to the genericity of the approach described by Jouault, Drivalos et al. propose domain-specific traceability metamodels for richer traceability link semantics [Drivalos *et al.* 2008]. Further research is required to assess the requirements of automated model synchronisation tools and to select appropriate traceability approaches for their implementation.

Impact analysis is used to reason about the effects of a change to a development artefact. As well as facilitating change propagation, impact analysis can help to predict the cost and complexity of changes [Bohner 2002]. Impact analysis requires solutions to several sub-problems, which include change detection, analysis of the effects of a change, and effective display of the analysis.

Impact analysis can be performed for UML models by comparing original and evolved versions of the same model to produce a report of evolved model elements that have been impacted by the changes to the original model elements [Briand *et al.* 2003]. To facilitate the impact analysis, Briand et al. identify change patterns that comprise, among other properties, a trigger (for change detection) and an impact rule (for marking model elements affected by this change). Figure 3.4 shows a sample impact analysis pattern for UML sequence diagrams, which is triggered when a message is added, and marks the sending class, the sending operation and the postcondition of the sending operation as impacted.

Only event-based approaches, such as the one described by Briand et al., have been proposed for automating impact analysis [Winkler & Pilgrim 2010]. Because of the use of patterns for detecting changes and specifying resulting actions, event-based impact analysis is similar to differencing approaches for schema evolution (for example, [Lerner 2000], which was discussed in Section 3.2.2). When more than one trigger might apply, event-based impact analysis approaches must provide mechanisms for selecting between applicable patterns. The selection policy used by Briand et al. is implicit (cannot be changed by the user) and does not provide a mechanism for selecting between applicable patterns.

Finally, model synchronisation tools might apply techniques used in automated synchronisation tools for traditional development environments. For example, the refactoring functionality of the Eclipse Java Development Tools [Fuhrer *et al.* 2007] propagates changes between classes using a cache of the workspace to improve performance and scalability.



**Change Title:** Changed Sequence Diagram – Added Message

**Change Code:** CSDVAM

**Changed Element:** `model::behaviouralElements::collaborations::SequenceDiagramView`

**Added Property:** `model::behaviouralElements::collaborations::Message`

**Impacted Elements:**
`model::foundation::core::ClassClassifier`
`model::foundation::core::Operation`
`model::foundation::core::Postcondition`

**Description:** The base class of the classifier role that sends the added message is impacted. The operation that sends the added message is impacted and its postcondition is also impacted.

**Rationale:** The sending/source class now sends a new message and one of its operations, actually sending the added message, is impacted. This operation is known or not, depending on whether the message triggering the added message corresponds to an invoked operation. If, for example, it is a signal then we may not know the operation, just by looking at the sequence diagram. The impacted postcondition may now not represent the effect (what is true on completion) of its operation.

**Resulting Changes:** The implementation of the base class may have to be modified. The method of the impacted operation may have to be modified. The impacted postcondition should be checked to ensure that it is still valid.

**Invoked Rule:** Changed Class Operation – Changed Postcondition (CCOCPst)

**OCL Expressions:**

```
context modelChanges::Change def:
    let addedMessage:Message = self.changedElement.oclAsType(SequenceDiagramView).
        Message->select(m:Message | m.getIDStr()=self.propertyID)
    let sendingOperation:Operation = (
        if addedMessage.activator.action.oclIsTypeOf(CallAction) then
            addedMessage.sender.base.operation->select(o:Operation |
            o.equals(addedMessage.activator.callAction.operation))
        else
            null
        endif)
context modelChanges::Change – class
    addedMessage.sender.base
context modelChanges::Change – operation
    sendingOperation
context modelChanges::Change – postcondition
    sendingOperation.postcondition
```

Figure 3.4: Example impact analysis pattern, taken from [Briand *et al.* 2003].

**Synchronisation of models with text and trace links** So far, this section has concentrated on model-to-model synchronisation, which is facilitated by traceability. Traceability is important for other software evolution activities in a model-driven development environment – such as synchronisation between models and text and between models and trace links – and these activities are now discussed.

While most of the model synchronisation literature focuses on synchronising models with other models, some papers consider synchronisation between models and other types of artefact. For synchronising changes in requirements documents with models, there is abundance of work in the field of requirements engineering, where the need for traceability was first reported. For synchronising models with generated text (during code generation, for example), the model-to-text language, Epsilon Generation Language (EGL) [Rose *et al.* 2008b], produces traceability links between code generation templates and generated files. Sections of code can be marked protected, and are not overwritten by subsequent invocations of the code generation template. The MOFScript model-to-text language, like EGL, provides protected sections. Unlike EGL, MOFScript stores traceability links in a structured manner, facilitating impact analysis, model coverage (for highlighting which areas of the model contribute to the generated code) and orphan analysis (for detecting invalid traceability links) [Olsen & Oldevik 2007].

Trace links can be affected when development artefacts change. Synchronisation tools rely on accurate trace links and hence the maintenance of trace links is important. It has been suggested that trace versioning should be used to address the challenges of trace link maintenance [Winkler & Pilgrim 2010], which include the accidental inclusion of unintended dependencies as well as the exclusion of necessary dependencies. Furthermore, Winkler and von Pilgrim note that, although versioning traces has been explored in specialised areas (such as hypermedia [Nguyen *et al.* 2005]), there is no holistic approach for versioning traces.

### Model-metamodel Co-Evolution

A metamodel describes the structures and rules for a family of models. When a model uses the structures and adheres to the rules defined by a metamodel, the model is said to *conform* to the metamodel [Bézivin 2005]. A change to a metamodel might require changes to models to ensure the preservation of conformance. The process of evolving a metamodel and its models together to preserve conformance is termed *model-metamodel co-evolution* and is subsequently referred to as *co-evolution.* This section explores existing approaches to co-evolution, comparing them with work from the closely related areas of schema and grammar evolution approaches (Sections 3.2.2 and 3.2.3). A more thorough analysis of co-evolution approaches is conducted in Chapter 4.

**Co-evolution theory** A co-evolution process involves changing a metamodel and updating instance models to preserve conformance. Often, the two activities are considered separately, and the latter is termed *migration.* In this thesis, the term *migration strategy* is used to mean an algorithm that specifies migration. Sprinkle and Karsai were the first to identify the need for approaches that consider the specific requirements of co-evolution [Sprinkle & Karsai 2004]. In particular, Sprinkle and Karsai describe migration as distinct from – and as having unique challenges compared to – the more general activity of model-to-model transformation. The phrase “evolution, not revolution” has been used to highlight and emphasise that, during co-evolution, the difference between source and target metamodels is often small [Sprinkle 2003].

Understanding the situations in which co-evolution must be managed is important for formulating appropriate requirements for co-evolution tools. Migration is sometimes made unnecessary by evolving a metamodel such that the conformance of models is not affected (e.g. making only additive changes) [Herrmannsdoerfer *et al.* 2009a]. Co-evolution can be carried out by more than one person, and in some cases metamodel developers and model users might not communicate [Cicchetti *et al.* 2008]. Notwithstanding these observations, the co-evolution literature rarely reports on the ways in which co-evolution is managed in practice.

**Co-evolution patterns** Much of the co-evolution literature suggests that the migration process should vary depending on the type of metamodel changes made (for example [Gruschko *et al.* 2007, Cicchetti *et al.* 2008]). In particular, the co-evolution literature identifies two important classifications of metamodel changes that affect the way in which migration is performed. Depending on the type of metamodel change, migration might be unnecessary (*non-breaking* change), can be automated (*breaking and resolvable* change) and can be automated only when ambiguity is resolved by a developer (*breaking and non-resolvable* change) [Gruschko *et al.* 2007]. Metamodel changes can also be regarded as either *metamodel-independent* (observed in the evolution of more than one metamodel) or *metamodel-specific* (observed in the evolution of only one metamodel) [Herrmannsdoerfer *et al.* 2008a].

Further research is needed to identify categories of metamodel changes, because many automated co-evolution approaches are underpinned by the categorisations. Although it has been suggested that a large proportion of metamodel changes re-occur [Herrmannsdoerfer *et al.* 2008a], the study in which this claim was made considers only two metamodels, both taken from the same organisation. Assessing the extent to which changes re-occur across a larger and broader range of metamodels is an open research challenge to which this thesis contributes, particularly in Chapter 4.

**Co-evolution approaches** Several approaches for managing co-evolution have been proposed, most of which are based on one of the two classifications of metamodel changes described above.

Re-use of migration knowledge is a primary concern in the work of Herrmannsdörfer, which is premised on an observation that a large proportion of metamodel changes re-occurred during the development of two metamodels [Herrmannsdoerfer *et al.* 2008a]. COPE [Herrmannsdoerfer *et al.* 2009a] is a co-evolution tool that provides a library of co-evolutionary operators. Operators are applied to evolve a metamodel and have predefined migration semantics. The application of each operator is recorded, and used to generate an executable migration strategy. Due to its use of re-usable operators, COPE shares characteristics with operator-based approaches for schema and grammar evolution (Sections 3.2.2 and 3.2.3). Consequently, the limitations of operator-based schema evolution approaches [Lerner 2000] apply to COPE. Balancing expressiveness and understandability is a key challenge for operator-based approaches because the former implies a large number of operators while the latter a small number of operators.

Gruschko et al. suggest inferring co-evolution strategies, based on either a difference model of two versions of the evolving metamodel (direct comparison) or on a list of changes recorded during the evolution of a metamodel (indirect comparison) [Gruschko *et al.* 2007]. To this end, Gruschko et al contribute the co-evolution process shown in Figure 3.5.

Two inference approaches inspired by the work of Gruschko et al. are now de-

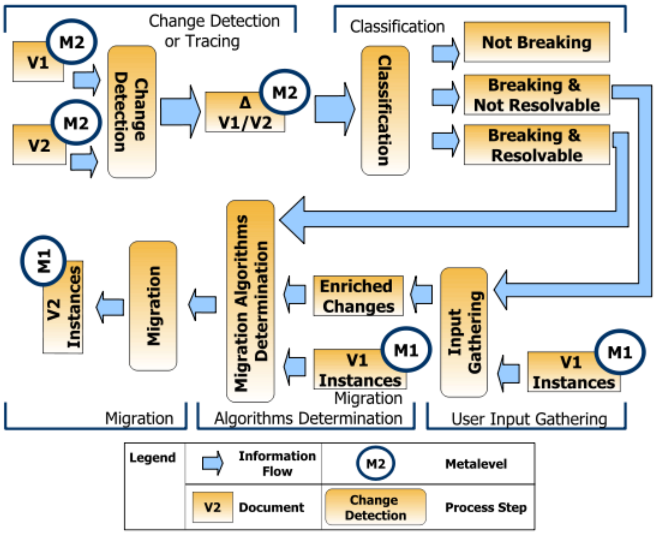


Figure 3.5: The co-evolution process described in [Gruschko *et al.* 2007].

scribed. Both approaches use a co-evolution process similar to the one shown in Figure 3.5, and use higher-order model transformation[5] for determining the migration strategy (the penultimate phase in Figure 3.5). Cicchetti et al. contribute a metamodel for describing the similarities and differences between two versions of a metamodel, enabling a model-driven approach to generating model migration strategies [Cicchetti *et al.* 2008]. Garcés et al. provide a similar metamodel, but use a metamodel matching process that can be customised by the user, who specifies matching heuristics to form a matching strategy [Garcés *et al.* 2009]. Otherwise, the co-evolution approaches described by Cicchetti et al. and Garcés et al. are fully automatic and cannot be guided by the user. Clearly then, accuracy is important for approaches that compare two metamodel versions, but Cicchetti et al. and Garcés et al. do not explore the extent to which the proposed approaches can be applied.

Some co-evolution approaches predate the classifications of metamodel changes described above. For instance, an initial catalogue of metamodel changes was the first to employ higher-order transformation for specifying model migration [Wachsmuth 2007]. However, Wachsmuth considers a small number of metamodel changes occurring in isolation and, as such, it is not clear whether the approach can be used in the general case. Sprinkle proposes a visual transformation language for specifying model migration based on graph transformation theory [Sprinkle 2003], which is less expressive than imperative or hybrid transformation languages (as discussed in Section 2.1.4).

---

[5] A model-to-model transformation that consumes or produces a model-to-model transformation is *higher-order*.

### Visualisation

To better understand the effects of evolution on development artefacts, visualising different versions of each artefact may be beneficial. Existing research for comparing text can be enhanced to perform semantic-differencing of models with a textual concrete syntax. For models with a visual concrete syntax, another approach is required.

To visualise the way in which model elements are transformed by transformation chains (the sequential composition of model-to-model transformations), a three-dimensional editor has been implemented [Pilgrim *et al.* 2008]. Figure 3.6 depicts a sample transformation chain visualisation. Each plane represents a model. The links between each plane illustrates the effects of a model-to-model transformation. The visualisation style used in the three-dimensional editor could be used to facilitate exploration of artefact evolution.

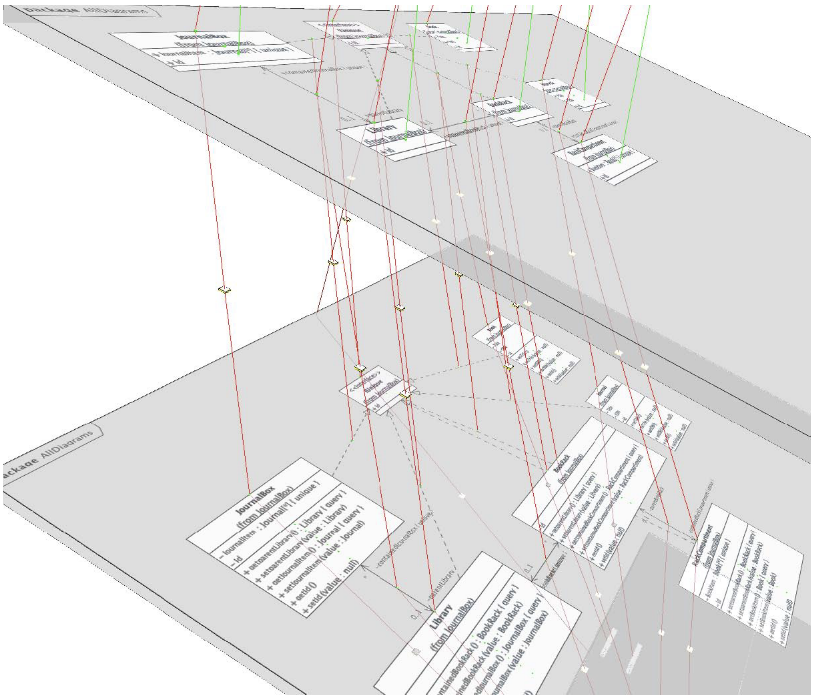


Figure 3.6: Visualising a transformation chain. Taken from [Pilgrim *et al.* 2008].

**Summary** Fully automated migration remains an open research challenge. Co-evolution approaches are in their infancy, and key problems need to be addressed. For example, matching schemas (metamodels) can yield more than one feasible set of migration strategies [Lerner 2000]. To this end, matching heuristics, which guide the inference of the model migration strategy – but might affect the predictability of the co-evolution process – have been proposed [Garcés *et al.* 2009].

Another open research challenge is in identifying an appropriate notation for describing migration. Existing tools have used general-purpose programming languages [Herrmannsdoerfer *et al.* 2009a] or higher-order model-to-model (M2M) transformations [Wachsmuth 2007, Cicchetti *et al.* 2008]. Migration is a specialisation of M2M transformation [Sprinkle & Karsai 2004], and therefore languages other than M2M transformation languages might be more suitable for describing migration.

Until co-evolution tools reach maturity, improving MDE modelling frameworks to better support co-evolution is necessary. For example, EMF (Section 2.3.1) cannot load models that no longer conform to their metamodel and, hence non-conformant models cannot be loaded in model editors, and cannot be used with model management operations.

This section has surveyed the existing work on identifying and managing software evolution in the context of MDE. In particular, several types of evolutionary change to modelling artefacts have been identified. Chapter 4 provides a more detailed and critical analysis of the work described in this section. In particular, Section 4.2 compares different approaches to managing co-evolution using an example from an MDE project.

## 3.3 Challenges to Managing Software Evolution in MDE

From the review presented above, several research challenges for managing software evolution in the context of MDE are synthesised. The challenges summarised in this section were considered as possible research directions for the thesis research.

**Model refactoring challenges** The model refactoring literature proposes tools and techniques for improving the quality of existing models without affecting their functional behaviour. In traditional development environments, inter-artefact refactoring (in which changes span more than one development artefact) is often automated, but none of the model refactoring papers discussed in this chapter consider inter-model refactoring. In general, the refactoring literature covers several concerns, such as identification, validation and assessment (Section 3.1.3), but the model refactoring literature considers only the specification and application of refactoring. To better understand the costs and benefits of model refactoring, further model refactoring research must also explore the identification, validation and assessment of model refactorings.

**Model synchronisation challenges** Improved scalability is the primary motivation of most model synchronisation research, although it has been suggested that model synchronisation can be used to improve the maintainability of a system via modularisation [Fritzsche *et al.* 2008]. Consequently, further research should explore the extent to which model synchronisation can be used to manage evolution. For impact analysis between models, only event-based approaches have been reported; other approaches – used successfully to manage evolution in other fields (such as relational databases and grammarware) – have not been applied to models [Winkler & Pilgrim 2010]. Few papers consider synchronisation with other artefacts and maintaining trace links and there is potential for further research in these areas.

**Co-evolution challenges** To better understand and to more thoroughly investigate model-metamodel co-evolution, further studies of the ways in which metamodels change are required. The existing empirical study of metamodel evolution in industry [Herrmannsdoerfer *et al.* 2008a] focuses only on two metamodels produced in the same organisation. Challenges for co-evolution reported in other fields have not been addressed by the model-metamodel co-evolution literature. For example, comparing two versions of a changed artefact (such as metamodel) can suggest more than one feasible migration strategy [Lerner 2000]. This challenge must be addressed by approaches to co-evolution that do not consider the way in which a metamodel has changed (e.g. [Cicchetti *et al.* 2008, Garcés *et al.* 2009]). A range of notations are used for model migration, including M2M transformation languages and general-purpose programming languages, which is a challenge for the comparison of co-evolution tools. Finally, contemporary MDE modelling frameworks do not facilitate MDE for non-conformant models, which is problematic at least until co-evolution tools reach maturity.

**General challenges for evolution in MDE** From the analysis in this chapter, several research challenges for software evolution in the context of MDE are apparent. Greater understanding of the situations in which evolution occurs informs the identification and management of evolution, yet few papers study evolution in real-world MDE projects. Analysis of existing projects can yield patterns of evolution, providing a common vocabulary for thinking and communicating about evolution. These patterns are used as a foundation for notations and tools used to automate some evolution activities. In addition, recording, analysing and visualising changes made over the long term to MDE development artefacts and to MDE projects is an area that is not considered in the literature.

## 3.4 Chapter Summary

This chapter has reviewed and analysed software evolution literature. Two evolution activities explored in the remainder of this thesis, *impact analysis* and *change propagation*, were introduced. Principles and practices of software evolution (from the fields of programming languages, relational database systems and grammarware) were compared and analysed. In particular, software evolution literature from the MDE community was reviewed and analysed to allow the synthesis of research challenges and potential directions for the thesis research.

As well as directing the thesis research, the literature review influenced the choice of research method (Section 1.4.2). Most of the software evolution research discussed uses a similar method (e.g. [Guerrini *et al.* 2005, Kramer 2001, Su *et al.* 2001] for XML schema evolution and [Banerjee *et al.* 1987, Edelweiss & Moreira 2005] for relational database schema evolution): first, identify and categorise evolutionary changes by considering all of the ways in which artefacts can change. Next, design a taxonomy of operators that capture these changes or a matching algorithm that detects the application of the changes. Then, implement a tool for applying operators, invoking a matching algorithm, or trigger change events. Finally, evaluate the tool on existing projects containing examples of evolution. This method assumes that most (if not all) evolutionary changes can be identified and captured from existing examples of evolution, and does not consider the ways in which evolution is already managed in existing projects. An alternative method was used to conduct the thesis research, which was discussed in Section 1.4.2 and based on the method used by Dig in his work on program refactoring ([Dig 2007]). First, existing projects were analysed to better understand the situations in which evolution occurs. From this analysis, research requirements were derived, and structures and processes for managing evolution were implemented. The structures and processes were evaluated by comparison with related structures and processes, and by application to real-world MDE projects in which evolution had occurred. This method was preferred as existing work on evolution in MDE (Section 3.2.4) does not consider the way in which evolution is managed in existing MDE projects, and, with the exception of COPE [Herrmannsdoerfer *et al.* 2009a], can only be used to manage a fixed set of evolutionary changes.

The review and the research challenges presented in this chapter led to a critical analysis of techniques for identifying and managing evolution in the context of software evolution. Chapter 4 presents the critical analysis, which was conducted using examples from real-world MDE projects.

# Chapter 4

# MDE and Evolution: Problem Analysis

The review presented in Chapter 3 highlighted challenges for identifying and managing evolution in the context of MDE, and noted that little work has explored the way in which evolution occurs in practice. This chapter explores evolution in the context of MDE by identifying and analysing examples from software engineering projects developed in a model-driven manner. The research method, outlined in Sections 1.4.2 and 3.4, comprises three phases: analysis, implementation and evaluation. This chapter describes the first phase, and presents the requirements that were identified for the implementation and evaluation phases.

Figure 4.1 summarises the work presented in this chapter. Examples of evolution in MDE projects were located from real-world MDE projects and additional sources (Section 4.1). The examples of evolution were used to analyse existing co-evolution techniques, which led to a categorisation and comparison of existing co-evolution approaches (Section 4.2) and to the identification of modelling framework characteristics that restrict the way in which co-evolution can be managed (Section 4.2.1). Research requirements for this thesis were identified from the analysis presented in this chapter (Section 4.3).

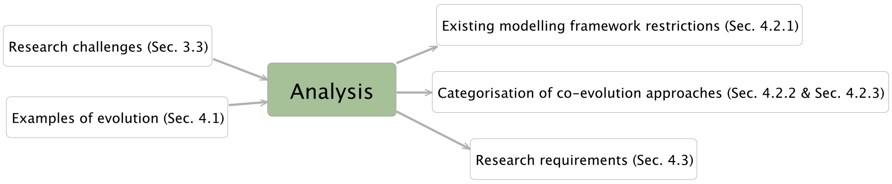


Figure 4.1: Analysis chapter overview.

## 4.1 Examples of Evolution from MDE Projects

In Chapter 3, three categories of evolutionary change were identified: model refactoring, synchronisation and co-evolution. Existing MDE projects were examined for examples of synchronisation and co-evolution and, due to time constraints, examples of model refactoring were not considered. The examples were used to provide requirements for developing structures and processes for evolutionary changes in the context of MDE. In this section, the requirements used to select example data are described, along with candidate and selected MDE projects. The section concludes with a discussion of further examples, which were obtained from joint research – with colleagues in this department and at the University of Kent – and from related work on the evolution of object-oriented programs.

### 4.1.1 Requirements

The suitability of each MDE project for analysing co-evolution and for analysing model synchronisation was determined using a set of requirements. The requirements were partitioned into: common requirements (needed to analyse both model synchronisation and co-evolution), requirements for studying model co-evolution, and requirements for studying synchronisation. Candidate MDE projects were evaluated against these requirements, and several were selected for further analysis.

#### Common requirements

Every project needs to use MDE (requirement R1), as the analysis investigated evolution in the context of MDE. Metamodelling and model transformation are two fundamental activities in MDE, as discussed in Section 2.1, and, as such, a project was deemed as using MDE when evidence of both metamodelling and model transformation was encountered.

Every project needs to provide historical information of development artefacts (R2), as the analysis investigated the evolution of development artefacts. For example, several versions of a project are needed, perhaps stored in a source code management system.

Every project needs to have undergone significant evolutionary change[1] (R3). Some types of evolution have little impact on other artefacts (adding a non-mandatory attribute to a class, for example), are not costly, and, hence, do not require dedicated structures and processes for their management.

#### Co-evolution requirements

Every project for the study of co-evolution needs to define a metamodel and some changes to that metamodel (R4), as co-evolution is the process of evolving a model in response to metamodel changes. In the projects considered, the metamodel changes

---

[1] This is deliberately vague. Further details are given in Section 4.1.2.

took the form of either another version of the metamodel, or a history (which recorded each of the steps used to produce the adapted metamodel)

Every project also needs to provide example instances of models before and after each migration activity (R5), as the way in which a model should be evolved during co-evolution is not always apparent from the metamodel changes. When a class is deleted from a metamodel, for example, models might be evolved such that all instances of the deleted class are also deleted, or such that all instances of the deleted class have a different type.

Ideally, every project will include more than one metamodel change in sequence, so as to represent the way in which the same development artefacts continue to evolve over time (optional requirement O1).

#### Synchronisation requirements

Every project for the study of synchronisation needs to define a model-to-model transformation (R6), as model synchronisation is the process of propagating changes between two models, and typically uses a model-to-model transformation to define the relationship between the two models.

Every project needs to provide examples of the kinds of change (to either source or target model) that cause inconsistency between the models (R7), so as to investigate the way in which models have changed in the context of the project (rather than investigate changes that might not have occurred in practice).

Ideally, every project will include transformation chains (more than one model-to-model transformation, executed sequentially) (O2), because chains of transformations are prescribed by the MDA guidelines [Kleppe *et al.* 2003], and hence model synchronisation tools might be required to propagate changes over a transformation chain (rather than over a single transformation).

### 4.1.2 Project Selection

Eight candidate projects were considered for the analysis presented in this chapter. Table 4.1 shows which of the requirements are met by each of the candidates. Each candidate is now discussed in turn.

#### GSN

The Goal Structuring Notation (GSN) [Kelly 1999] is a notation for specifying safety arguments. Georgios Despotou and Tim Kelly, members of this department's High Integrity Systems Engineering group, are constructing a metamodel for Goal Structuring Notation (GSN). The metamodel has been developed incrementally. There is no accurate and detailed version history for the GSN metamodel (requirement R2). **Suitability for study:** Unsuitable.

| Name | Requirements | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Common | | | Co-evolution | | | Synchronisation | | |
| | R1 | R2 | R3 | R4 | R5 | O1 | R6 | R7 | O2 |
| GSN | x | | | x | | | | | |
| OMG | x | | | x | | | x | | |
| Zoos | x | x | | x | | | | | |
| MDT | x | x | | x | | x | | | |
| MODELPLEX | x | x | x | x | | x | x | | |
| FPTC | x | x | x | x | x | | | | |
| Xtext | x | x | x | x | x | x | x | | x |
| GMF | x | x | x | x | x | x | x | | x |

Table 4.1: Candidates for study of evolution in existing MDE projects

### OMG MDE Projects

The Object Management Group (OMG)[2] oversees the development of model-driven technologies. Andrew Watson, the Vice President and Technical Director of OMG, oversaw the development of two MDE projects [Watson 2008]. Personal correspondence with Watson ascertained that source code is available for one of the projects, but there is no version history. **Suitability for study:** Unsuitable.

### Zoos

A zoo is a collection of metamodels, authored in a common metamodelling language. Two zoos were considered (the Atlantic Zoo and the AtlantEcore Zoo[3]), but neither contained significant metamodel changes. Those changes that were recorded involved only renaming of meta-classes (trivial to migrate) or additive changes, which do not affect conformance and therefore require no migration. **Suitability for study:** Unsuitable.

### MDT

The Eclipse Model Development Tools (MDT)[4] provides implementations of industry-standard metamodels, such as UML2 [OMG 2007a] and OCL [OMG 2006]. Like the metamodel zoos, the version history for the MDT metamodels contained no significant changes. **Suitability for study:** Unsuitable.

---

[2] `http://www.omg.org`

[3] Both have moved to: `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`

[4] `http://www.eclipse.org/mdt`

### MODELPLEX

Jendrik Johannes, a Research Assistant at TU Dresden, has made available work from the European project, MODELPLEX[5]. Johannes's work involves transforming UML models to Tool Independent Performance Models (TIPM) for simulation. Although the TIPM metamodel and the UML-to-TIPM transformation have been changed significantly, no significant changes have been made to the models. The TIPM metamodel was changed such that conformance was not affected. **Suitability for study:** Unsuitable.

### FPTC

Failure Propagation and Transformation Calculus (FPTC) [Wallace 2005], developed by Malcolm Wallace in this department, provides a means for reasoning about the failure behaviour of complex systems. In an earlier project, Richard Paige and the author developed an implementation of FPTC in Eclipse [Paige *et al.* 2009]. The implementation includes an FPTC metamodel. More recent work with Philippa Conmy, a Research Associate in this department, has identified a significant flaw in the implementation, leading to changes to the metamodel. The metamodel changes affected the conformance of existing FPTC models. Conmy has made available copies of FPTC models from before and after the changes. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation, because, although the tool includes a transformation, the target models are produced as output from a simulation, never stored and hence do not become inconsistent with their source model.

### Xtext

Xtext[6] is used to generate parsers, metamodels and editors for performing text-to-model transformation, and was introduced in Section 2.1.4. Internally, Xtext defines a metamodel, which was changed significantly between 2006 and 2008. In several cases, changes have affected conformance. Xtext provides examples, which have been updated alongside the metamodel. Xtext uses model-to-model transformation to generate models, which are later disregarded and hence do not need to be synchronised with source models. **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation.

### GMF

The Graphical Modelling Framework (GMF) [Gronback 2009] allows the definition of graphical concrete syntax for metamodels that have been defined in EMF, and was introduced in Section 2.3.1. GMF prescribes a model-driven approach: users of GMF define concrete syntax as a model, which is used to generate a graphical editor. In fact, five models are used together to define a single editor using GMF.

---

[5] Grant number IST 34081, `http://www.modelplex.org/`

[6] `http://www.eclipse.org/Xtext/`

GMF defines the metamodels for graphical, tooling and mapping definition models; and for generator models. The metamodels have changed considerably during the development of GMF. Some changes have affected the conformance of existing GMF models. Presently, migration is encoded in Java. Gronback has stated[7] that the migration code is being ported to QVT (a model-to-model transformation language) as the Java code is difficult to maintain.

GMF meets almost all of the requirements for the study. Co-evolution data is available, including migration strategies. The GMF source code repository does not contain examples of the kinds of change that cause inconsistency between the models (R7). **Suitability for study:** Suitable for studying co-evolution. Unsuitable for studying synchronisation.

#### Summary of selection

Of the eight projects considered, three (FPTC, Xtext and GMF) met all of the requirements for studying co-evolution. No project met all of the requirements for studying synchronisation, and, consequently, data was sought from other sources as discussed below. FPTC and Xtext were used to perform the analysis described in the remainder of this chapter, along with examples taken from other sources. GMF provides the most comprehensive examples of co-evolution, as it includes several metamodels that have undergone two major and several minor revisions, several exemplar models that have been migrated, and reference migration strategies (written in Java). Rather than use GMF for analysis, it was instead reserved for evaluation of the thesis research (Chapter 6).

### 4.1.3 Other examples

Since few existing MDE projects met all of the requirements for studying evolution, additional data was sought from alternative sources. Examples were located from object-oriented systems – which have some similarities to systems developed using MDE – and via collaboration with colleagues on two projects, both of which involved developing a system using MDE.

#### Examples of evolution from object-oriented software

Object-oriented programming and metamodelling have some commonalities. In object-oriented programming, software is constructed by developing groups of related objects. Every object is an instance of (at least) one class. A class is a description of characteristics, which is shared by each of the class's instances (objects). A similar relationship exists between models and metamodels: metamodels comprise meta-classes, which describe the characteristics shared by each of the meta-class's instances (elements of a model). Together, model elements are used to describe one perspective (model) of a system. This similarity between object-oriented programming and metamodelling implied

---

[7] Private communication, 2008.

that the evolution of object-oriented systems may be similar to evolution occurring in MDE.

*Refactoring* is the process of improving the structure of existing code while maintaining its external behaviour. When used as a noun, a refactoring is one such improvement. As discussed in Chapter 3, refactoring of object-oriented systems has been widely studied, perhaps most notably in [Fowler 1999], which provides a catalogue of refactorings for object-oriented systems. For each refactoring, Fowler gives advice and instructions for its application.

To explore their relevance to MDE, Fowler's refactorings have been applied to metamodels. Some have been found to be relevant to metamodels, and could potential occur during MDE. Many have been found to be irrelevant, belonging to one of the following three categories:

1. **Operational refactorings** focus on restructuring behaviour (method bodies). Most modelling frameworks do not support the specification of behaviour in models.
2. **Navigational refactorings** affect navigational constructs are specified, such as changing between bi- and uni-directional associations. These changes are often non-breaking in modelling frameworks, which typically infer values for the inverse of a reference when required.
3. **Domain-specific refactorings** manage issues not relevant to metamodels, such as casting, defensive return values, and assertions.

The object-oriented refactorings that can be applied to metamodels provide examples of metamodel evolution and, in some cases, have the potential to affect conformance. For each refactoring that affected conformance, a migration strategy was deduced by the author using Fowler's description of each refactoring. An example of this process is now presented.

Figure 4.2 illustrates a refactoring that changes a reference object to a value object [Fowler 1999][pg183]. Value objects are immutable, and cannot be shared (i.e. any two objects cannot refer to the same value object). By contrast, reference objects are mutable, and can be shared. Figure 4.2 indicates that applying the refactoring restricts the multiplicity of the association (on the Order end) to 1 (implied by the composition); prior to the refactoring the multiplicity is many-valued.

Before applying the refactoring, each customer may be associated with more than one order. After the refactoring, each customer should be associated with only one order. Fowler indicates that every customer associated with more than one order should be duplicated, such that one customer object exists for each order. Therefore, the migration strategy in Listing 4.1 is deduced. Using this process, migration strategies were deduced for each of the refactorings that were applicable to metamodels and affected conformance.

```
for every customer, c
  for every order, o, associated with c
```

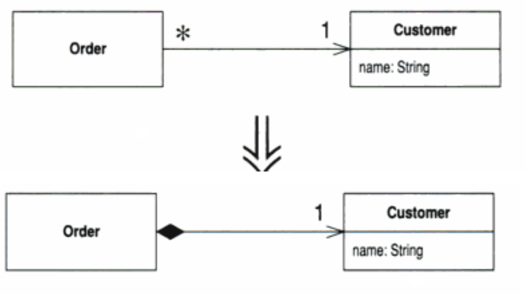


Figure 4.2: Refactoring a reference to a value. Taken from [Fowler 1999, pg183].

```
    create a new customer, d
    copy the values of c's attributes into d
  next o

  delete c
next c
```

Listing 4.1: Pseudo code migration strategy for the refactoring in Figure 4.2

The examples of metamodel evolution based on Fowler's refactorings provided additional data for deriving research requirements. Some parts of the metamodel evolutions from existing MDE projects were later found to be equivalent to Fowler's refactorings, which, to some extent, validates the above claim that evolution from object-oriented systems can be used to reason about metamodel evolution.

However, object-oriented refactorings are used to improve the maintainability of existing systems. In other words, they represent only one of the three common reasons for evolutionary change (Section 3.1.1). The two other types of change – for addressing new requirements and facilitating interoperability with other systems – are equally relevant for deriving research requirements, and so object-oriented refactorings alone are not sufficient for reasoning about metamodel evolution.

### Research collaborations

As well as the example data located from object-oriented systems, collaboration on projects using MDE with two colleagues provided several examples of evolution. A graphical editor for process-oriented programs was developed with Adam Sampson, then a Research Associate at the University of Kent, and is described in Appendix B. Additionally, the feasibility of a tool for generating story-worlds for interactive narratives was investigated with Heather Barber, then a postdoctoral researcher in this department.

In both cases, a metamodel was constructed for describing concepts in the domain. The metamodels were developed incrementally and changed over time. The collaborations with Sampson and Barber did not involve constructing model-to-model transformations, but did provide data suitable for a study of co-evolution.

The majority of the changes made in both of these projects relate to changing requirements. In each iteration, existing requirements were refined and new requirements discovered. Neither project required changes to support architectural restructuring. In addition, the work undertaken with Sampson included some changes to adapt the system for use with a different technology than originally anticipated. That is to say, the changes observed represented two of the three common reasons for evolutionary change (Section 3.1.1).

### 4.1.4 Summary

This section has described the identification of examples for analysing the way in which evolution is identified and managed in the context of MDE. Example data was sought from existing MDE projects, a related domain (refactoring of object-oriented systems) and collaborative work on MDE projects (with Sampson and Barber). Eight MDE projects were located, three of which satisfied the requirements for a study of co-evolutionary changes in the context of model-driven engineering. One of the three projects, GMF, was reserved for the evaluation presented in Chapter 6. Refactorings of object-oriented programming supplemented the data available from the existing MDE projects. Collaboration with Sampson and Barber yielded further examples of co-evolution.

The lack of examples of model synchronisation might indicate that model synchronisation is not an important evolutionary activity for MDE. An alternative explanation is that, in 2008, many projects were beginning to adopt MDE principles, but had not transitioned to a fully model-driven style of development. It has been suggested that MDE is adopted in an incremental manner and that model transformations are specified only after models are first used in an ad-hoc manner and MDE is applied to automate simple tasks [Rios *et al.* 2006]. Regular, empirical assessment is necessary to investigate the adoption of MDE and the need to support evolutionary activities, such as model synchronisation. Due to the lack of examples of model synchronisation, the remainder of the thesis focuses on model-metamodel co-evolution.

## 4.2 An Analysis of Existing Co-Evolution Techniques

The examples of co-evolution identified in the Section 4.1 were analysed to identify and compare existing techniques for managing co-evolution. This section discusses the results of analysing the examples; namely a deeper understanding of modelling framework characteristics that affect the management of co-evolution (Section 4.2.1), and a categorisation of existing techniques for managing co-evolution (Sections 4.2.2 and 4.2.3). The work presented here was published in [Rose *et al.* 2009b, Rose *et al.* 2009a].

### 4.2.1 Modelling Framework Characteristics

The co-evolution examples identified in Section 4.1 were examined to understand the ways in which co-evolution is managed in practice. The examination of co-evolution examples highlighted characteristics of MDE development environments that affect the way in which co-evolution can be managed. The characteristics are now described.

#### Model-Metamodel Separation

In MDE development environments, *models and metamodels are separated.* Metamodels are developed and distributed to users. Metamodels are installed, configured and combined to form a customised MDE development environment. Metamodel developers have no programmatic access to instance models, which reside in a different workspace and potentially on a different machine. Consequently, metamodel evolution occurs independently to model migration. Figure 4.3 shows the activities typically involved in co-evolution. First, the metamodel developer evolves the metamodel and may create a migration strategy. Subsequently, the metamodel users discover conformance problems after installing the new version of the metamodel, and migrate their models.

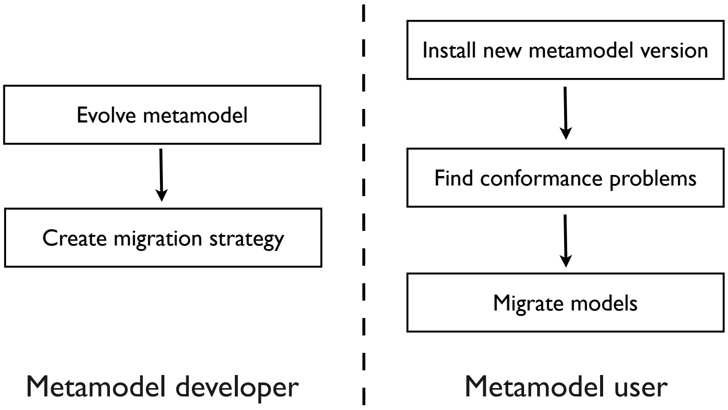


Figure 4.3: Co-evolution activities

Due to model and metamodel separation, co-evolution is either *developer-driven* (the metamodel developer devises an executable migration strategy, which is distributed to the metamodel user with the evolved metamodel) or *user-driven* (the metamodel developer provides no migration strategy). In either case, model migration occurs on the machine of the metamodel user, after and independent of metamodel evolution.

#### Implicit Conformance

MDE development environments *implicitly enforce conformance.* A model is *bound* to its metamodel, typically by constructing a representation in the underlying programming language for each model element and data value. Frequently, binding is strongly-typed: each metamodel type is mapped to a corresponding type in the underlying programming language using mappings defined by the metamodel. Consequently, modelling frameworks do not permit changes to a model that would cause it to no longer conform to its metamodel. Loading a model that does not conform to its metamodel causes an error. In short, MDE modelling frameworks cannot be used to manage models that do not conform to their metamodel.

Because modelling frameworks can only load models that conform to their metamodel, user-driven co-evolution is always a manual process, in which models are migrated without using the modelling framework. Typically then, the metamodel user can only perform migration by editing the model directly, normally manipulating its underlying representation (e.g. XMI). Model editors and model management operations, which are ordinarily integral to MDE, cannot be used to manage models that do not conform to their metamodel and hence, cannot be used during model migration.

A further consequence of implicitly enforced conformance is that modelling tools must produce models that conform to their metamodel, and therefore, model migration cannot be decomposed. Consequently, model migration cannot be performed by combining co-evolution techniques, because intermediate steps must produce conformant models.

### 4.2.2 User-Driven Co-Evolution

Examples of co-evolution were analysed to discover and compare existing techniques for managing co-evolution. As discussed above, the separation of models and metamodels leads to two processes for co-evolution: *developer-driven* and *user-driven*. Analysis of the co-evolution examples identified in Section 4.1 highlighted several instances of user-driven co-evolution. Projects conducted in collaboration with Barber and with Sampson involved user-driven co-evolution, and all of the co-evolution examples taken from the Xtext project were managed in a user-driven manner. This section demonstrates user-driven co-evolution using a scenario similar to one observed during the collaboration with Barber.

In user-driven co-evolution, the metamodel user performs migration by loading their models to test conformance, and then reconciling conformance problems by updating non-conformant models. The metamodel developer might guide migration by providing a migration strategy to the metamodel user. Crucially, however, the migration strategy is not executable (e.g. it is written in prose). This is the key distinction between user-driven and developer-driven co-evolution. Only in the latter does the metamodel developer provided an executable model migration strategy.

In some cases, the metamodel user will not be provided with any migration strategy (executable or otherwise) from the metamodel developer. To perform migration, the metamodel user must determine which (if any) model elements no longer conform to the evolved metamodel, and then decide how best to change non-conformant elements to re-establish conformance.

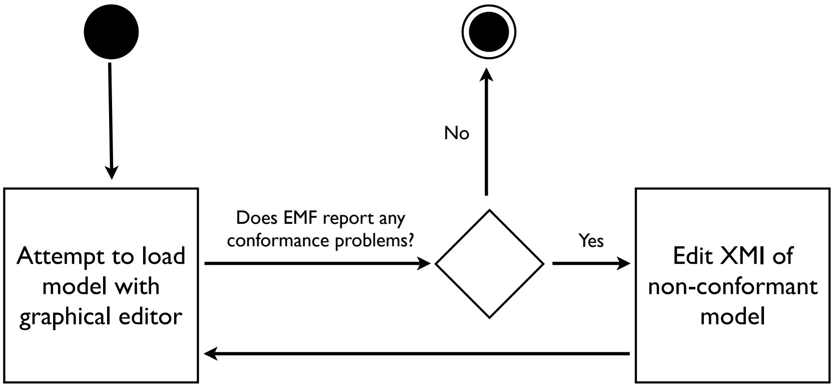


Figure 4.4: User-driven co-evolution with EMF

### User-driven co-evolution in EMF

EMF (Section 2.3.1) provides structures and processes for loading, storing and editing models and metamodels. When a model no longer conforms to its metamodel, however, the model cannot be loaded, cannot be used with EMF's model editors, and cannot be used with model management operations. Therefore, model migration must be performed manually, typically by editing the underlying storage representation of the model.

A typical workflow for performing user-driven co-evolution with EMF is shown in Figure 4.4. (The workflow in Figure 4.4 assumes a graphical model editor, such as those generated by GMF, but any model editor that is built on EMF will exhibit the same behaviour). When the user attempts to load a model, EMF automatically checks the conformance of the model with respect to its metamodel. When the model does not conform to its metamodel, EMF reports (some of the) conformance errors, loading fails and the model is not displayed in the graphical editor. To re-establish conformance, the user must edit by hand the underlying storage representation of the model, XMI. After saving the reconciled XMI to disk, the user attempts to load the model in the graphical editor again. If the user makes a mistake in reconciling the XMI, loading will fail again and further conformance errors will be reported. Even if the user makes no mistakes in reconciling the XMI, further conformance errors might be reported because EMF uses a multi-pass XMI parser and cannot report all categories of conformance problem in one pass of the XMI. If further conformance problems are reported, the user continues to reconcile the XMI by hand. Otherwise, migration is complete and the model is displayed in the graphical editor.

**Scenario** The following scenario demonstrates user-driven co-evolution. Mark is developing a metamodel. Members of his team, including Heather, install Mark's metamodel and begin constructing models. Mark later identifies new requirements, changes the metamodel, builds a new version of the metamodel, and distributes it to his colleagues.

After several iterations of metamodel updates, Heather tries to load one of her older models, constructed using an earlier version of Mark's metamodel. When loading the older model, the modelling framework reports an error indicating that the model no longer conforms to its metamodel. To load the older model, Heather must reinstall the version of the metamodel to which the older model conforms. But even then, the modelling framework will bind the older model to the old version of the metamodel, and not to the evolved metamodel.

Employing user-driven co-evolution, Heather must trace and repair the loading error directly in the model as it is stored on disk. Model storage formats have typically been optimised to either reduce the size of models on disk or to improve the speed of random access to model elements. Therefore, human usability is not a key requirement for model storage formats. XMI, for example, is a standard model storage format and is regarded as sub-optimal for use by humans [OMG 2004]. Consequently, using a model storage format to perform model migration can be error-prone and tedious. When directly editing the underlying format of a model, reconciling conformance is often a slow and iterative process. With EMF [Steinberg *et al.* 2008], for example, user-driven co-evolution is an iterative process as shown in Figure 4.4. When models are persisted in a database (perhaps for scalability reasons), a binary rather than a textual storage representation is used, further impeding user-driven co-evolution. For example, models stored using the Connected Data Objects Model Repository (CDO)[8] are persisted in a relational database, which must be manipulated when non-conformant models are to be edited.

### Challenges

The scenario (in the box above) highlights the two most significant challenges faced when performing user-driven co-evolution. Firstly, the underlying model representation is unlikely to be optimised for human usability and hence user-driven co-evolution is error-prone and tedious. Secondly, although conformance can be affected when a new version of a metamodel is installed, conformance problems are not reported to the user as part of the installation process. These challenges are further elaborated in the Section 4.3, which identifies research requirements.

It is worth noting that the above scenario describes a metamodel with only one user. Some metamodels – such as UML, Ecore, and MOF – have many more users, and user-driven co-evolution would require repeated manual effort from each user. In spite of this, UML, for example, does not provide a strategy for migrating between versions

---

[8] `http://www.eclipse.org/cdo/`

of the specification, and users must infer the migration semantics from changes to the specification.

### 4.2.3 Developer-Driven Co-Evolution Approaches

In developer-driven co-evolution, the metamodel developer provides an executable migration strategy along with the evolved metamodel. Model migration might be scheduled automatically by the modelling framework (for example when a model is loaded) or by the metamodel user.

As noted in Section 4.2.2, co-evolution research focuses on developer-driven rather than user-driven co-evolution. Several developer-driven co-evolution approaches were reviewed in Section 3.2.4. To compare and categorise existing developer-driven co-evolution approaches, the approaches have been applied by the author to the co-evolution examples identified in Section 4.1. From this analysis and from the literature review conducted in Section 3.2.4, three categories of developer-driven co-evolution approach were identified: *manual specification*, *operator-based* and *inference*. The categorisation has been published in [Rose *et al.* 2009b]. Each category is now discussed.

#### Manual Specification

In *manual specification*, the migration strategy is encoded manually by the metamodel developer, typically using a general purpose programming language (e.g. Java) or one of the model-to-model transformation languages described in Section 2.1.4. The migration strategy can manipulate instances of the metamodel in any way permitted by the modelling framework. Manual specification approaches have been used to manage migration in GMF (Section 2.3.1) and in the Eclipse MDT UML2 project, which were outlined in Section 4.1. Compared to operator-based and inference techniques (below), manual specification permits the metamodel developer the most control over model migration.

However, manual specification generally requires the most effort on the part of the metamodel developer for two reasons. Firstly, as well as implementing the migration strategy, the metamodel developer must also produce code for executing the migration strategy. Typically, this involves integration of the migration strategy with the modelling framework (to load and store models) and possibly with development tools (to provide a user interface). Secondly, frequently occurring model migration patterns – such as copying a model element from original to migrated model – are not captured by existing general purpose and model-to-model transformation languages, and so each metamodel developer has to codify migration patterns in their chosen language.

#### Operator-based

In *operator-based co-evolution* techniques, a library of *co-evolutionary operators* is provided. Each co-evolutionary operator specifies a metamodel evolution along with a corresponding model migration strategy. For example, the “Make Reference Containment” operator might evolve the metamodel such that a non-containment reference becomes a containment reference and migrate models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code. Operator-based approaches for schema evolution and co-evolution were described in Sections 3.2.2 and 3.2.4. In particular, a library of co-evolutionary operators for MOF metamodels has been described in [Wachsmuth 2007], and COPE [Herrmannsdoerfer *et al.* 2009a] is an operator-based co-evolution approach for EMF.

The efficacy of an operator-based co-evolution approach depends heavily on the richness of its library of co-evolutionary operators. When no operator describes the required co-evolution pattern, the metamodel developer must use another approach for performing model migration. For instance, COPE allows migration to be specified manually with a general purpose programming language when no co-evolutionary operator is appropriate. (Consequently, custom migration strategies in COPE suffer one of the same limitations as manual specification approaches: model migration patterns are not captured in the language used to specify migration strategies).

As using co-evolutionary operators to express migration require the metamodel developer to write no code, it seems that operator-based co-evolution approaches should seek to provide a large library of co-evolutionary operators, so that at least one operator is appropriate for every co-evolution pattern that a metamodel developer may wish to apply. However, as discussed in Section 3.2.2, a large library of operators increases the complexity of specifying migration. To demonstrate, Lerner considers moving a feature from one type to another. This could be expressed by sequential application of two operators called, for example, `delete_feature` and `add_feature`. However, the semantics of a `delete_feature` operator are likely to dictate that the values of that feature will be removed during migration and hence, `delete_feature` is unsuitable when specifying that a feature has been moved. To solve this problem, a `move_feature` operator could be introduced, but then the metamodel developer must understand the difference between the two ways in which moving a type can be achieved, and carefully select the correct one. Lerner provides other examples which further elucidate this issue (such as introducing a new type by splitting an existing type). As the size of the library of co-evolutionary operators grows, so does the complexity of selecting appropriate operators and, hence, the complexity of performing metamodel evolution.

Clear communication of the effects of each co-evolutionary operator (on both the metamodel and its instance models) can improve the navigability of large libraries of co-evolutionary operators. COPE, for example, provides a name, description, list of parameters and applicability constraints for each co-evolutionary operator. An example is shown in Figure 4.5. To choose between operators, users can read descriptions (such as the one shown below) examine the source code of the operator, or try executing the operator (an undo command is provided).

Finding a balance between richness and navigability is a key challenge in defining libraries of co-evolutionary operators for operation-based co-evolution approaches. Analogously, a known challenge in the design of software interfaces is the trade-off between a rich and a concise interface [Bloch 2005].

**Make Reference Containment**
In the metamodel, a reference is made [into a] containment. In the model, its values are replaced by copies.
*Parameters*:

- `reference`: The reference

*Constraints*:

- The `reference` must not already be containment.

Figure 4.5: Make Reference Containment operation, taken from COPE [Herrmannsdoerfer *et al.* 2009a].

To perform metamodel evolution using co-evolutionary operators, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE, for instance, provides integration with the EMF tree-based metamodel editor. However, some developers edit their metamodels using a textual syntax, such as Emfatic[9]. In general, freeform text editing is less restrictive than tree-based editing (because in the latter, the metamodel is always structurally sound whereas in the former, the text does not always have to compile). Consequently, it is not clear whether operator-based co-evolution can be used with all categories of metamodel editing tool.

### Inference

In *inference* approaches, a migration strategy is derived from the evolved metamodel and the *metamodel history.* Inference approaches can be further categorised according to the type of metamodel history used. *Differencing* approaches compare and match the original and evolved metamodels, while *change recording* approaches use a record of primitive changes made to the original metamodel to produce the evolved metamodel. The analysis of the evolved metamodel and the metamodel history yields a *difference model* [Cicchetti *et al.* 2008], a representation of the changes between original and evolved metamodel. The difference model is used to infer a migration strategy, typically by using a higher-order model-to-model transformation[10] to produce a model-to-model transformation from the difference model. Cicchetti et al. and Garcés et al. describe differencing-based inference approaches [Cicchetti *et al.* 2008, Garcés *et al.* 2009]. There exist no pure change recording approaches, although COPE [Herrmannsdoerfer *et al.* 2009a] uses change recording for the specification of custom

---

[9] `http://www.alphaworks.ibm.com/tech/emfatic`

[10] A model-to-model transformation that consumes or produces a model-to-model transformation is termed a higher-order model transformation.

model migration strategies, and Méndez et al. suggest that a change recording approach might be used to manage metamodel-transformation co-evolution [Méndez *et al.* 2010].

Compared to manual specification and operator-based co-evolution approaches, inference approaches require the least amount of effort from the metamodel developer who needs only to evolve the metamodel and provide a metamodel history. However, for some types of metamodel change, there is more than one feasible model migration strategy. For example, when a metaclass is deleted, one feasible migration strategy is to delete all instances of the deleted metaclass. Alternatively, the type of each instance of the deleted metaclass could be changed to another metaclass that specifies equivalent structural features.

To select the most appropriate migration strategy from all feasible alternatives, an inference approach often requires guidance, because the metamodel changes alone do not provide enough information to correctly distinguish between feasible migration strategies. Existing inference approaches use heuristics to determine the most appropriate migration strategy. These heuristics sometimes lead to the selection of the wrong migration strategy.

Because inference approaches use heuristics to select a migration strategy, it can sometimes be difficult to reason about which migration strategy will be selected. For domains where predictability, completeness and correctness are a primary concern (e.g. safety critical or security critical systems, or systems that must undergo certification with respect to a relevant standard), such approaches are unsuitable, and deterministic approaches that can be demonstrated to produce correct, predictable results will be required.

The two types of inference approach – differencing and change recording – are now compared, using an example of co-evolution, introduced below.

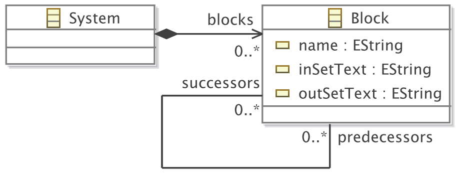
**Example** The following example was observed during the development of the Epsilon FPTC tool [Paige *et al.* 2009] (which was summarised in Section 4.1). The source code is available from EpsilonLabs[11]. Figure 4.6(a) illustrates the original metamodel in which a `System` comprises any number of `Blocks`. A `Block` has a name, and any number of `successor` `Blocks`; `predecessors` is the inverse of the `successors` reference.

Further analysis of the domain revealed that extra information about the relationship between `Blocks` was to be captured. The evolved metamodel is shown in Figure 4.6(b). The `Connection` class is introduced to capture the extra information via its `literalsText` attribute. `Blocks` are no longer related directly to `Blocks`, instead they are related via an instance of the `Connection` class. The `incomingConnections` and `outgoingConnections` references of `Block` are used to relate `Blocks` to each other via an instance of `Connection`.

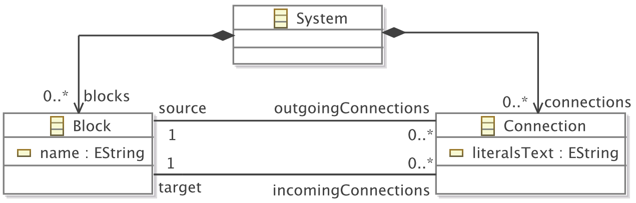
A model that conforms to the original metamodel (Figure 4.6(a)) might not conform to the evolved metamodel (Figure 4.6(b)). Below is a description of the strategy used

---

[11] `http://sourceforge.net/projects/epsilonlabs/`



(a) Original metamodel, prior to evolution



(b) Evolved metamodel with Connection metaclass

Figure 4.6: Metamodel evolution in the Epsilon FPTC tool. Taken from [Rose *et al.* 2009b].

by the Epsilon FPTC tool to migrate a model from original to evolved metamodel and is taken from [Rose *et al.* 2009b]:

```
1. For every instance, b, of Block:
      For every successor, s, of b:
         Create a new instance, c, of Connection.
         Set b as the source of c.
         Set s as the target of c.
         Add c to the connections of the System containing b.
2. And nothing else changes.
```

Using the example described above, differencing and change recording inference approaches are now compared.

**Change recording** In change recording approaches, metamodel evolution is monitored by a tool, which records a list of primitive changes (e.g. Add class named `Connection`, Change the type of feature `successors` from `Block` to `Connection`). The record of changes may be reduced to a normal form to remove redundancy, but doing so can erase useful information. In change recording, some types of metamodel evolution can be more easily recognised than with differencing. With change recording, renaming can be distinguished from a deletion followed by an addition. With differencing, this distinction is not possible.

In general, more than one combination of primitive changes can be used to achieve the same metamodel evolution. However, when recording changes, the way in which a metamodel is evolved affects the inference of migration strategy. In the example presented above, the `outgoingConnections` reference (shown in Figure 4.6(b)) could have been produced by changing the name and type of the `successors` reference (shown in Figure 4.6(a)). In this case, the record of changes would indicate that the new `outgoingConnections` reference is an evolution of the `successors` reference, and consequently an inferred migration strategy would be likely to migrate values of `successors` to values of `outgoingConnections`. Alternatively, the metamodel developer may have elected to delete the `successors` reference and then create the `outgoingConnections` reference afresh. In this record of changes, it is less obvious that the migration strategy should attempt to migrate values of `successors` to values of `outgoingConnections`. Clearly then, change recording approaches require the metamodel developer to consider the way in which their metamodel changes will be interpreted.

Change recording approaches require facilities for monitoring metamodel changes from the metamodel editing tool, and from the underlying modelling framework. As with operation-based co-evolution, it is not clear to what extent change recording can be supported when a textual syntax is used to evolve a metamodel. A further challenge is that the granularity of the metamodel changes that can be monitored influences the inference of the migration strategy, but this granularity is likely to be controlled by and specific to the implementation of the metamodelling language. Normal forms to which a record of changes can be reduced have been proposed to address this issue [Cicchetti 2008].

**Differencing** In differencing approaches, the original and evolved metamodels are compared to produce the difference model. Unlike change recording, metamodel evolution may be performed using any metamodel editor; there is no need to monitor the primitive changes made to perform the metamodel evolution. However, as discussed above, not recording the primitive changes can cause some categories of change to become indistinguishable, such as renaming versus a deletion followed by an addition.

To illustrate this problem further, consider again the metamodel evolution described above. A comparison of the original (Figure 4.6(a)) and evolved (Figure 4.6(b)) metamodels shows that the references named `successors` and `predecessors` no longer exist on `Block`. However, two other references, named `outgoingConnections` and `incomingConnections`, are now present on `Block`. A differencing approach might deduce (correctly, in this case) that the two new references are evolutions of the old references. However, no differencing approach is able to determine which mapping is correct from the following two possibilities:

- `successors` evolved to `incomingConnections`; `predecessors` evolved to `outgoingConnections`.

- `successors` evolved to `outgoingConnections`; `predecessors` evolved to `incomingConnections`.

The choice between these two possibilities can only made by the metamodel developer, who knows that `successors` (`predecessors`) is semantically equivalent to `outgoingConnections` (`incomingConnections`). As shown by this example, fully automatic differencing approaches cannot always infer a migration strategy that will capture the semantics desired by the metamodel developer.

### 4.2.4 Summary

Analysis of existing co-evolution techniques has led to a deeper understanding of modelling frameworks characteristics that are relevant for co-evolution, to the identification of user-driven co-evolution and to a categorisation of developer-driven co-evolution techniques.

Modelling frameworks separate models and metamodels and, hence, co-evolution is a two-step process. To facilitate model migration, metamodel developers may codify an executable migration strategy and distribute it along with the evolved metamodel (developer-driven co-evolution). When no executable migration strategy is provided, models must be migrated by hand (user-driven co-evolution). Because modelling frameworks implicitly enforce conformance, user-driven co-evolution is performed by editing the underlying storage representation of models, which is error-prone and tedious.

User-driven co-evolution, which has not been explored in the literature, was observed in several of the co-evolution examples discussed in Section 4.1. In situations where the metamodel developer has not specified or cannot specify an executable migration strategy, user-driven co-evolution is required.

Existing techniques for performing developer-driven co-evolution have been compared and categorised. The categorisation highlights a trade-off between flexibility and effort for the metamodel developer when choosing between categories of approach, as shown in Figure 4.7.

Manual specification affords the metamodel developer more flexibility in the specification of the migration strategy, but, because languages that do not capture reoccurring model migration patterns are typically used, may require more effort. By contrast, inference approaches derive a migration strategy from a metamodel history and hence require less effort from the metamodel developer. However, an inference approach affords the metamodel developer less flexibility, and may restrict the metamodel evolution process because, for example, the order of metamodel changes affects the inference of a migration strategy. Operator-based approaches occupy the middle-ground: by restricting the way in which metamodel evolution is expressed, an operator-based approach can be used to infer a migration strategy. The metamodel developer selects appropriate operators that express both metamodel evolution and model migration. Operator-based approaches require a specialised metamodel editor, and it is not yet

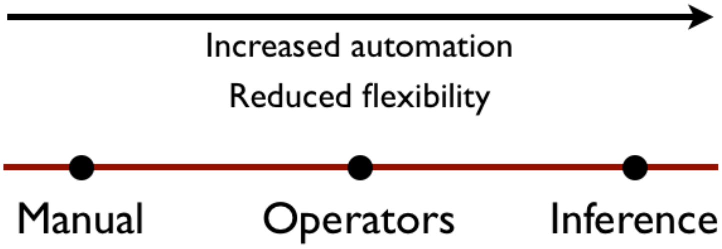


Figure 4.7: Spectrum of developer-driven co-evolution approaches

clear whether they can be applied when a metamodel is represented with a freeform (e.g. textual) rather than a structured (e.g. tree-based) syntax.

## 4.3 Requirements for Identifying and Managing Co-Evolution

The analysis presented throughout this chapter has highlighted a number of challenges for identifying and managing model-metamodel co-evolution. Several factors affect and restrict the way in which co-evolution is performed in practice. The way in which modelling frameworks are implemented affect the ways in which the impact analysis and propagation of metamodel changes can be performed. Existing co-evolution approaches are developer-driven rather than user-driven (i.e. assume that the metamodel developer will provide a migration strategy), which was not the case for several of the examples identified in Section 4.1. Additionally, the languages used to specify model migration vary over existing co-evolution approaches, which inhibits the conceptual and practical re-use of model migration patterns. This section contributes requirements for structures and processes that seek to address these challenges.

Below, the thesis requirements are presented in three parts. The first identifies requirements that seek to extend and enhance support for managing model and metamodel co-evolution with modelling frameworks. The second summarises and identifies requirements for enhancing the user-driven co-evolution process discussed in Section 4.2.2. Finally, the third identifies requirements that seek to improve the spectrum of existing developer-driven co-evolution techniques.

### 4.3.1 Explicit conformance checking

Section 4.2.1 discussed characteristics of modelling frameworks relevant to managing co-evolution. Because modelling frameworks typically enforce model and metamodel conformance implicitly, they cannot be used to load non-conformant models. Consequently, user-driven co-evolution involves editing a model in its storage representation, which is error-prone and tedious, because human usability is not normally a key requirement for model storage representations. Furthermore, modelling frameworks that implicitly enforce conformance understandably provide little support for explicitly checking the conformance of a model with other metamodels (or other versions of the same metamodel). As discussed in Section 4.2.1, explicit conformance checking is useful for impact analysis activities, such as determining whether a model needs to be migrated during the installation of a newer version of its metamodel.

Therefore, the following requirement was derived: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

### 4.3.2 User-driven co-evolution

When a metamodel change will affect conformance in only a small number of models, a metamodel developer may decide that the extra effort required to specify an executable migration strategy is too great, and prefer a user-driven co-evolution technique. Section 4.2.2 introduced – and highlighted several challenges for – user-driven co-evolution.

Because modelling frameworks typically cannot be used to load models that do not conform to their metamodel, user-driven co-evolution involves editing the storage representation of a model. As discussed above, model storage representations are typically not optimised for human use and hence user-driven co-evolution can be error-prone and time consuming. When a multi-pass parser is used to load models (as is the case with EMF), user-driven co-evolution is an iterative process, because not all conformance errors are reported at once.

Therefore, the following requirement was derived: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a conformance report for the original model and evolved metamodel.*

### 4.3.3 Developer-driven co-evolution

The comparison of developer-driven co-evolution techniques (Section 4.2.3) highlights variation in the languages used for codifying model migration strategies. More specifically, the model migration strategy languages varied in their scope (general-purpose programming languages versus model transformation languages) and category of type system. Furthermore, the amount of processing performed when executing a model migration strategy also varied: some techniques only load a model, execute the model migration strategy using an existing execution engine and store the model, while others perform significant processing in addition to the computation specified in the model migration strategy. COPE, for example, transforms models to a metamodel-independent representation before migration is executed, and back to a metamodel-specific representation afterwards.

Of the three categories of developer-driven co-evolution technique identified in Section 4.2.3, only manual specification (in which the metamodel developer specifies the migration strategy by hand) always requires the use of a migration strategy language. Nevertheless, both operator-based and inference approaches might utilise a migration strategy language in particular circumstances. Some operator-based approaches, such as COPE, permit manual specification of a model migration strategy when no co-evolutionary operator is appropriate. For describing the effects of co-evolutionary operators, the model migration part of an operator could be described using a model migration strategy language. When an inference approach suggests more than one feasible migration strategy, a migration strategy language could be used to present alternatives to the metamodel developer. To some extent then, the choice of model migration strategy language influences the efficacy of all categories of developer-driven co-evolution approach.

Given the variations in existing model migration strategy languages and the influence of those languages on developer-driven co-evolution, the following requirement was derived: *This thesis must compare and evaluate existing languages for specifying model migration strategies.*

As discussed in Section 4.2.3, existing manual specification techniques do not provide model migration strategy languages that capture patterns specific to model migration. Developers must re-invent solutions to commonly occurring model migration patterns, such as copying an element from the original to the migrated model. In some cases, manual specification techniques require the developer to implement, in addition to a migration strategy, mechanisms for loading and storing models and for interfacing with the metamodel user.

Devising a domain-specific languages or DSL (discussed in Chapter 3) is one approach to capturing re-occurring patterns, and executable DSLs are widely used for specifying and performing model management in contemporary model-driven development environments. Contemporary model management tools provide many executable DSLs for model transformation, validation, merging, weaving, and many other model management tasks (Section 2.1.4).

Given the apparent appropriateness of a domain-specific language for specifying model migration and that no common language for specifying migration has yet been devised, the following requirement was derived: *This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies, comparing it to existing languages for specifying model migration strategies.*

## 4.4 Chapter Summary

The literature review performed in Chapter 3 identified several types of evolution that occur in MDE projects, including model refactoring, synchronisation and co-evolution. Although several papers propose structures and processes for managing evolution in MDE, little work has considered the way in which MDE artefacts evolve in practice. The work described in this chapter has investigated evolution in existing MDE projects, culminating in a deeper understanding of the conceptual and technical issues faced when identifying and managing co-evolution. Furthermore, the analysis has facilitated the derivation of requirements for structures and processes that will address several of the challenges to identifying and managing co-evolution today.

Examples of co-evolution were identified from real-world MDE projects, and supplementary data was located by examining a related area (refactoring in object-oriented systems) and from collaborative work on two projects using MDE. The examples were used to understand how co-evolution is performed in practice, and led to the identification of user-driven co-evolution. Furthermore, the examples were used to analyse and categorise existing approaches to managing co-evolution.

Examining the co-evolution examples and applying existing co-evolution tools to the examples led to several observations. Firstly, modelling frameworks restrict the way in which co-evolution can be identified and managed. Secondly, user-driven co-evolution (in which models are migrated without an executable strategy) occurs in practice, but no existing co-evolution tools provide support for it. Finally, the variation of languages used for specifying model migration inhibits the re-use of commonly occurring patterns.

From the analysis performed in this chapter, requirements for the implementation phase of the thesis were formulated. The structures and process developed to approach those requirements are described in Chapter 5, and seek to alleviate the restrictions of modelling frameworks, to improve and support user-driven co-evolution (which is currently error-prone and tedious), and to provide a common language for specifying model migration.

# Chapter 5

# Design and Implementation

Section 4.3 presented requirements for structures and processes for identifying and managing co-evolution. This chapter describes the way in which the requirements have been addressed. Several related structures have been implemented, using domain-specific languages, metamodelling and model management operations. Figure 5.1 summarises the contents of the chapter. To facilitate the management of non-conformant models with existing modelling frameworks, a metamodel-independent syntax was devised and implemented (Section 5.1). To address some of the challenges faced in user-driven co-evolution, an OMG specification for a textual modelling notation was implemented (Section 5.2). Finally, a model transformation language – tailored for model migration and centred around a novel approach to relating source and target model elements – was designed and implemented (Sections 5.3 and 5.4).

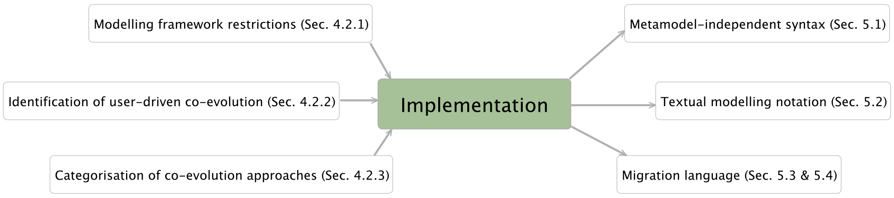


Figure 5.1: Implementation chapter overview.

The structures presented in this chapter are interoperable as shown in Figure 5.2. In particular, the modelling framework extensions of the metamodel-independent syntax are used to provide conformance checking for the textual modelling notation, and to enable conformance checking for the model migration language. The structures were separated to facilitate re-use of the conformance checking services provided by the metamodel-independent syntax. Table 5.1 shows the relationship between the proposed structures and the thesis requirements (Section 4.3).

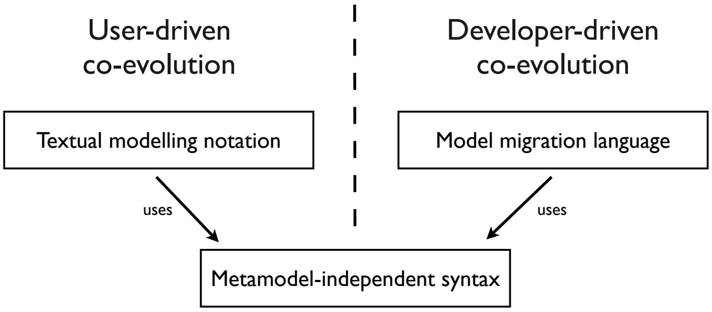


Figure 5.2: The relationships between the proposed structures

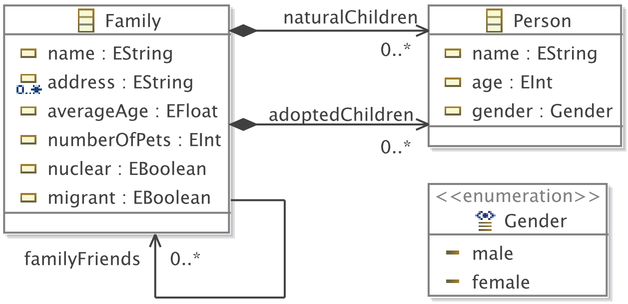
| **Structure (Section)** | **Requirement** |
|---|---|
| Metamodel-independent syntax (5.1) | This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels. |
| Textual modelling notation (5.2) | This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a conformance report for the original model and evolved metamodel. |
| Model migration language (5.3) | This thesis must compare and evaluate existing languages for specifying model migration strategies. |
| Model migration language (5.4) | This thesis must implement and evaluate a domain-specific language for specifying and executing model migration strategies, comparing it to existing languages for specifying model migration strategies. |

Table 5.1: The relationship between the thesis requirements and the proposed structures

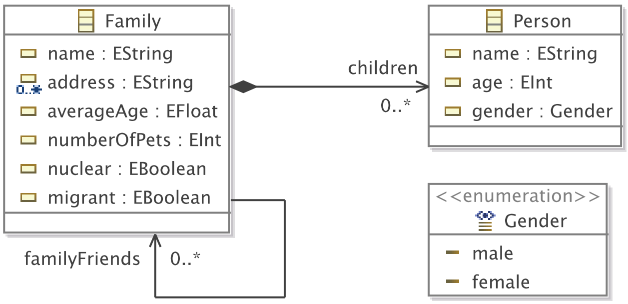
## 5.1 A Metamodel-Independent Syntax

Section 4.2.1 discussed the way in which modelling frameworks implicitly enforce conformance, and hence prevent the loading of non-conformant models. Additionally, modelling frameworks provide little support for checking the conformance of a model with other versions of a metamodel, which is potentially useful during metamodel installation. In Section 4.3, these concerns led to the identification of the following requirement: *This thesis must investigate the extension of existing modelling frameworks to support the loading of non-conformant models and conformance checking of models against other metamodels.*

This section describes the way in which existing modelling frameworks load and store models using metamodel-specific binding mechanisms, proposes an alternative binding mechanism using a metamodel-independent syntax, and demonstrates how this facilitates automatic consistency checking. The work presented in this section has been published in [Rose *et al.* 2009a].



(a) Original metamodel.



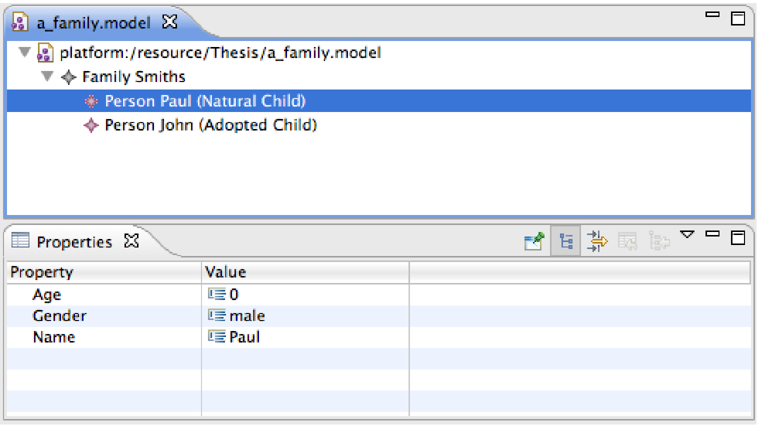
(b) Evolved metamodel.

Figure 5.3: Evolution of a families metamodel, based on the metamodel in [OMG 2004].

### 5.1.1 Metamodel Evolution Example: Families

This section uses the example of metamodel evolution in Figure 5.3. The metamodels in Figure 5.3 have been constructed in Ecore, the metamodelling language of EMF, which is based on MOF (Section 2.1.3). The metamodels use Ecore types such as `EString` and `EFloat`. The `nuclear` attribute on the `Family` type is used to indicate that the family "comprises only a father, a mother, and children" [Merriam-Webster 2010], and not extended family members (such as cousins or grandparents).

In Figure 5.3(a), `naturalChildren` and `adoptedChildren` are modelled as separate features, and, in Figure 5.3(b), they are modelled as a single feature, `children`. Models that specify values for the `naturalChildren` or `adoptedChildren` features do not conform to the evolved metamodel. For example, the model in Figure 5.4 represents a `Family` comprising two `Persons`, conforms to the original metamodel, and does not conform to the evolved metamodel. Using the families metamodel and model, the sequel explains why existing modelling frameworks cannot be used to load non-conformant models.



platform:/resource/Thesis/a_family.model
  Family Smiths
    Person Paul (Natural Child)
    Person John (Adopted Child)

| Property | Value |
|---|---|
| Age | 0 |
| Gender | male |
| Name | Paul |

Figure 5.4: A family model, which conforms to the metamodel in Figure 5.3(a), in an EMF tree editor

### 5.1.2 Binding to a Specific Metamodel

To load a model, existing modelling frameworks construct objects in the underlying programming language in a process termed *binding* (Section 4.2.1). The metamodel defines the way in which model elements will be bound, and binding is strongly-typed. Figure 5.5 illustrates the results of binding the family model in Figure 5.4 to the original families metamodel in Figure 5.3(a). The objects in Figure 5.5 instantiate types that are defined in the metamodel, such as `Family` and `Person`. In other words, binding results in a *metamodel-specific* representation of the model.

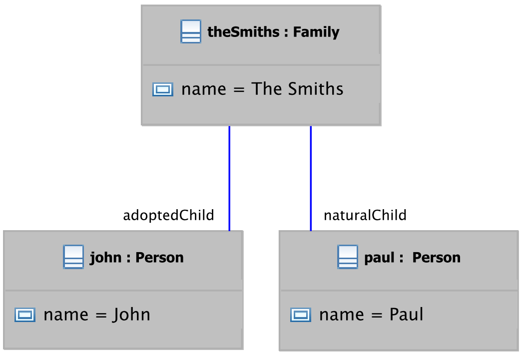


Figure 5.5: Objects resulting from the binding of a conformant model, as a UML object diagram

Metamodel-specific binding fails for non-conformant models. For example, attempting to bind the family model in Figure 5.4 to the evolved families metamodel (Figure 5.3(b)) fails because the model uses `naturalChildren` and `adoptedChildren` features for the type `Family`, and these features are not defined by the evolved metamodel.

Because non-conformant models cannot be loaded, model migration must be performed by editing the underlying storage representation, which can be error-prone and tedious (Section 4.2.2). The sequel discusses potential solutions for loading non-conformant models.

### 5.1.3 Potential Solutions for Loading Non-Conformant Models

Two potential approaches to binding (and hence loading) non-conformant models have been considered and are now discussed. The benefits and drawbacks of each approach have been compared, which resulted in the selection of the second approach, binding to a metamodel-independent syntax.

#### Store metamodel history

Presently, modelling frameworks are used to store only the latest version of a metamodel, and hence binding fails for models that conform to a previous version of the metamodel. If modelling frameworks could access old versions of a metamodel, models that do not conform to the current version of the metamodel could be loaded by binding to a previous version of the metamodel.

#### A metamodel-independent syntax

Models can always be successfully bound to a *metamodel-independent* representation, such as the one shown in Figure 5.6. Binding each model element results in the instantiation of a metamodel-independent type (`Object` in Figure 5.6) rather than of types defined in a specific metamodel, such as `Family` or `Person`. Hence, binding is independent of the types defined in metamodels, and will succeed for non-conformant models.

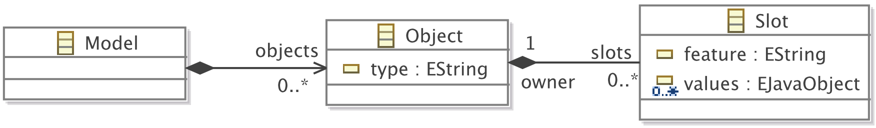


Figure 5.6: A minimal generic metamodel for MOF in Ecore, based on [OMG 2008a] and taken from [Rose *et al.* 2009a].

### Benefits and drawbacks of the potential solutions

The two potential solutions for loading non-conformant models have different benefits and drawbacks, which are now discussed. Storing metamodel histories would use the binding and conformance checking services provided by existing modelling frameworks, and therefore require less implementation effort than a metamodel-independent syntax, which would require bespoke binding and conformance checking services. Furthermore, structures for managing metamodel histories might be integrated with existing approaches to managing co-evolution, such as metamodel differencing approaches (Sections 4.2.3), for switching between different versions of a MDE workflow.

Storing metamodel histories relies on the metamodel developer to enable model migration: if the metamodel developer does not provide a metamodel that contains historical data, then binding will fail for non-conformant models. Conversely, models can be bound to a metamodel-independent syntax irrespective of the actions of the metamodel developer.

A metamodel-independent syntax has been chosen because it makes fewer assumptions of the metamodel developer, and hence facilitates user-driven as well as developer-driven co-evolution.

## 5.1.4 Proposed Solution: A Metamodel-Independent Syntax

This section discusses the design and implementation of a metamodel-independent syntax, and of the binding and conformance checking services that are used to load non-conformant models. As discussed below, the metamodel-independent syntax and conformance checking service are inspired by UML [OMG 2007b] and [Paige *et al.* 2007], respectively. As such, the primary contribution of this section is the implementation and integration of the syntax and services with EMF. In addition, the syntax and services have been designed to be re-usable, and hence have been used to simplify the implementation of a textual modelling notation (Section 5.2) and a model migration language (Section 5.4).

### Design

A high-level design for the way in which the metamodel-independent syntax, binding service and conformance checking service load models is shown in Figure 5.7. The **binding service** parses XMI (the canonical storage representation of models, Section 2.1.3) and produces a model that conforms to the **metamodel-independent syntax**. The **conformance checking service** is used to explicitly check the conformance of a model conforming to the metamodel-independent syntax.

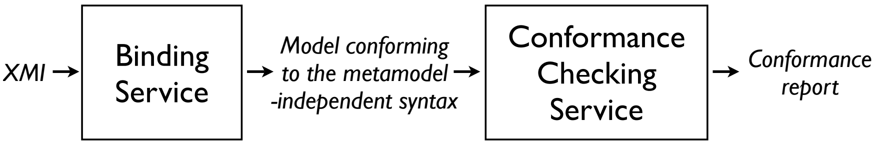


Figure 5.7: Loading models with the metamodel-independent syntax

Binding and conformance checking were split into separate services to facilitate re-use. For example, the textual modelling notation in Section 5.2 re-uses the metamodel-independent syntax and conformance checking service, in conjunction with a different binding service.

**Metamodel-independent syntax** The metamodel-independent syntax is used to represent a model without instantiating types defined by its metamodel. Its design was inspired by the metamodel for UML 2 [OMG 2007a] object diagrams, which describes objects in a generic, class-independent manner. UML 2 object diagrams are specified in terms of an abstract syntax (comprising, for example, `InstanceSpecification` and `Link` classes) and a concrete syntax (comprising, for example, boxes and lines). The metamodel-independent syntax proposed here is abstract. It is not used directly by metamodel developers or users and hence a concrete syntax was not required.

Abstract syntax is typically represented as a metamodel (Section 2.1.2). The metamodel in Figure 5.6 was used as an initial design for the metamodel-independent syntax, which contains a class for each type in the MOF metamodel that is instantiated in a model. In other words, `Objects` are used to represent each element of a model, and the `type` attribute is used to indicate the name of the metaclass that the `Object` intends to instantiate. Similarly, `Slots` are used to represent values in the model, and the `feature` attribute indicates the metafeature that the `Slot` intends to instantiate. The metamodel was designed to capture the information needed to perform conformance checking (described below), and implementing the conformance checking service led to a refactored metamodel, which is presented in the sequel.

COPE (Section 4.2.3) is underpinned by a metamodel-independent syntax. However, the metamodel-independent syntaxes used by COPE and proposed here were developed independently, and both were first published in 2008 (in [Rose *et al.* 2008a, Herrmannsdoerfer *et al.* 2008b]).

**Metamodel-independent binding service** The metamodel-independent binding service is a text-to-model (T2M) transformation that consumes XMI and produces a model conforming to the metamodel-independent syntax. The transformation was designed to extract all of the information pertaining to the model from XMI, translating it into the concepts defined in the metamodel-independent syntax. In particular, the binding service iterates over each tag in the XMI, and creates instances of `Object` and `Slot`. For example, when encountering a tag that represents a model element, the transformation performs the steps in Figure 5.8.

Applying the metamodel-independent binding service to the families model (Figure 5.4) produces three instances of `Object`, illustrated as a UML object diagram in Figure 5.9. For clarity, instances of `Object` are shaded, and instances of `Slot` are unshaded. The first `Object` represents the `Family` model element and has three slots. Two of the slots are used to reference the `Person` model elements via the `naturalChildren` and `adoptedChildren` references.

**Conformance checking service** Conformance is a type of inter-model consistency, between a model and its metamodel (Section 2.1.2), and, in MDE, inter-model consistency is often validated using a set of constraints (Section 2.1.4). Furthermore, conformance can be specified as a set of constraints between a model and its metamodel [Paige *et al.* 2007]. As such, the conformance checking service has been designed as the set of constraints between models and metamodels in Figure 5.11.

The conformance checking service must be interoperable with the metamodel-independent syntax and, hence, the constraints are specified in terms of `Objects` and `Slots`. Clearly, to check conformance the constraints must refer to a (specific) metamodel, and the constraints are also specified in terms of concepts from the MOF metamodelling language (Section 2.1.3), such as `Class` and `Property`. Figure 5.10 shows a minimal version of the MOF metamodel.

After binding to the metamodel-independent syntax, the conformance of a model can be checked against any specific metamodel. To illustrate the value of the conformance checking service, consider again the metamodel evolution in Figure 5.3 and the bound model in Figure 5.9. For the evolved metamodel (Figure 5.3(b)), conformance checking for the model element representing the `Family` would fail. As illustrated in Figure 5.9, the `Family Object` defines slots for features named `naturalChildren` and `adoptedChildren`, which are not defined the metaclass `Family` in Figure 5.3(b). Specifically, the model element representing the `Family` does not satisfy conformance constraint 3, which states: *each `Slot`'s feature must be the name of a metamodel `Property`. That `Property` must belong to the `Slot`'s owner's `type`.* The conformance checking service provides a report of conformance problems, which can be used during co-evolution by tools and users.

### Reference implementation in Java, EMF and Epsilon

Reference implementations of the metamodel-independent syntax, the binding service and the conformance service were constructed with Java, EMF and Epsilon (Sec-

```
1. Constructs an instance of Object, o.
2. For each attribute of the tag:
       Creates an instance of Slot, s.
       Sets s.feature to the name of the attribute.
       Sets s.value to the value of the attribute.
       Adds s to o.slots.
3. For each child tag:
       Creates an instance of Slot, s.
       Sets s.feature to the name of the child tag.
       Recursively constructs an instance of Object, c.
       Sets s.value to c.
       Adds s to o.slots.
```

Figure 5.8: Pseudo code for binding XMI tags to `Objects`.

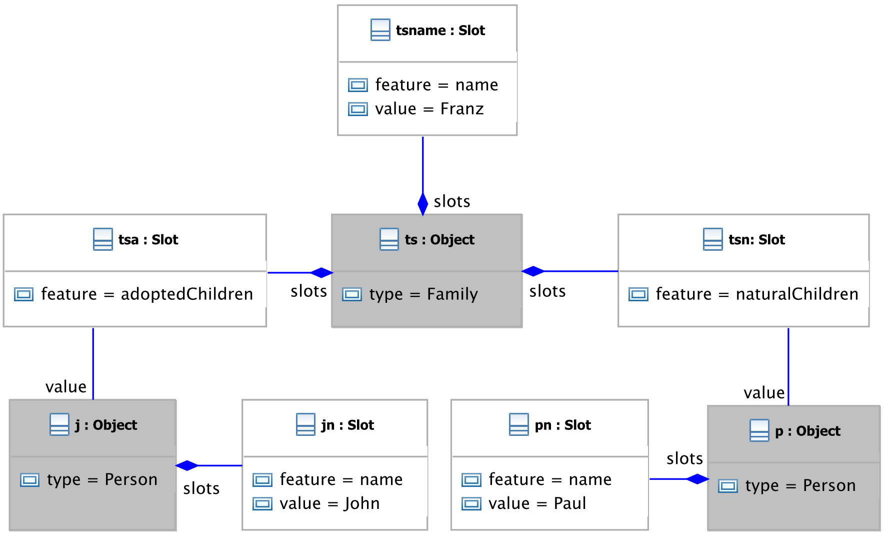


Figure 5.9: Result of binding the families model with the metamodel-independent syntax

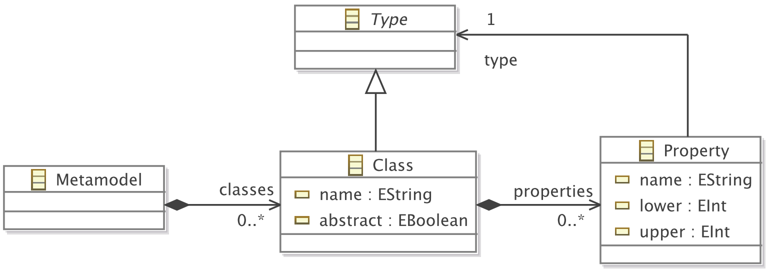


Figure 5.10: Minimal MOF metamodel, based on [OMG 2008a].

1. Each `Object`’s `type` must be the `name` of some non-`abstract` metamodel `Class`.
2. Each `Object` must specify a `Slot` for each mandatory `Property` of its `type`.
3. Each `Slot`’s `feature` must be the name of a metamodel `Property`. That `Property` must belong to the `Slot`’s `owner`’s `type`.
4. Each `Slot` must be multiplicity-compatible with its `Property`. More specifically, each `Slot` must contain at least as many values as its `Property`’s `lower` bound, and at most as many values as its `Property`’s `upper` bound.
5. Each `Slot` must be type-compatible with its `Property`. (The way in which type-compatibility is checked depends on the way in which the modelling framework is implemented).

Figure 5.11: The constraints of the conformance checking service.

tion 2.3). The way in which each component was implemented is now discussed.

**Metamodel-independent syntax** Ecore, the metamodelling language of EMF, was used to implement the metamodel-independent syntax. The final metamodel is shown in Figure 5.12, which differs slightly from the initial design (Figure 5.6). Specifically, `Slot` is abstract, has a generic type (`T`), and is the superclass of `AttributeSlot`, `ReferenceSlot` and `ContainmentSlot`. These changes simplified the implementation of the (abstract) `typeCompatibileWith` method, which is used by the conformance checking service, and returns `true` if and only if every element of the `values` attribute is type compatible with the `EClassifier` parameter (a metamodel type).

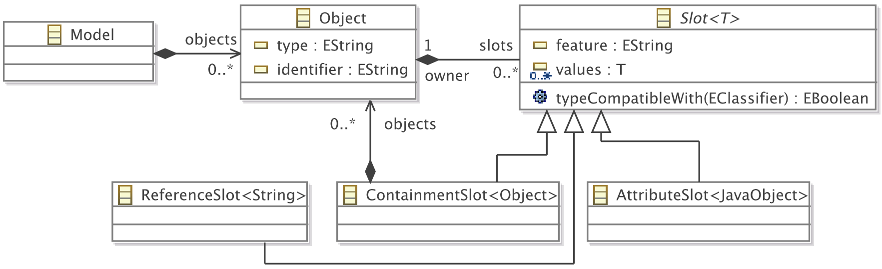


Figure 5.12: Implemented version of the metamodel-independent syntax, in Ecore

**Binding service** A text-to-model (T2M) transformation language (Section 2.1.4) could have been used to implement the binding service. However, when this work started (2008) the Eclipse Modeling Project[1] did not provide a standard T2M language and using a T2M language that was not part of the Eclipse Modeling Project would have complicated installation of the service for users.

Instead, the binding service has been implemented by constructing in Java an XMI parser that emits objects conforming to the metamodel-independent syntax. Listing 5.1 illustrates the way in which XMI attributes are parsed. The `processAttributes` method is called to generate instances of `AttributeSlot` from the metamodel-independent syntax (Figure 5.12). For each attribute in an XMI tag, the body of the loop is executed. If the attribute is not XMI metadata such as type information (line 4), the name and value of the attribute (lines 5 and 6) are extracted from the XMI, and used to add the value to an `AttributeSlot` with feature equal to the name of the attribute (line 8). Constructing `Objects` and `Slots` is the responsibility of the `generator` object, which is an instance variable of the parser.

```
1  private void processAttributes(Attributes atts) {
2    for (int index = 0; index < atts.getLength(); index++) {
3
```

---

[1] `http://www.eclipse.org/modeling/`

```
 1 context Object {
 2  constraint ClassMustNotBeAbstract {
 3   check: not self.toClass().isAbstract()
 4   message: ’Cannot instantiate the abstract class: ’ + self.type
 5  }
 6 }
 7
 8 operation Object toClass() : EClass {
 9  return Metamodel!EClass.all.selectOne(c|c.name == self.type);
10 }
```

Listing 5.2: A constraint (in EVL) to check that only concrete metamodel types are instantiated

```
    if (!attributeIsMetadata(atts.getQName(index))) {
      final String feature = atts.getLocalName(index);
      final String value = atts.getValue(index);

      generator.addAttributeValue(feature, value);
    }
  }
}
```

Listing 5.1: Parsing XMI attributes (in Java)

**Conformance checking service** EVL (Section 2.1.4), a language tailored for model verification and hence suitable for rapid prototyping of consistency constraints, was used to implement the conformance constraints (Figure 5.11). Listing 5.2 shows the EVL constraint that checks whether each `Object`’s `type` is a non-abstract class (constraint 1 in Figure 5.11). The `check` part (line 3) verifies that a particular `Object` (referenced via the `self` keyword) refers to a metamodel type that is not abstract. If the check fails, the message (line 4) is automatically added to a set of unsatisfied constraints. The `toClass` operation (lines 8-10) is used to determine the metamodel class (an instance of `EClass`) to which the `type` attribute (a `String`) of an `Object` refers. The conformance checking service returns a report of unsatisfied constraints.

Type-compatibility checks have been implemented using the type-checking methods provided by EMF. The EVL constraints call the `isTypeCompatibleWith` method on the `Slot` class. Each subclass of `Slot` provides an implementation of `isTypeCompatibleWith`, which delegates to EMF to perform type-checking.

### 5.1.5 Applications of the Metamodel-Independent Syntax

There are many potential uses for the metamodel-independent syntax described in this section. Section 5.2 describes a textual modelling notation integrated with the metamodel-independent syntax to achieve live conformance checking. In addition to this, the metamodel-independent syntax is potentially useful during metamodel installation. As discussed in Section 4.2.1, metamodel developers do not have access to downstream models, and conformance is implicitly enforced by modelling frameworks. Consequently, the conformance of models may be affected by the installation of a new version of a metamodel, and the conformance of models cannot be checked during installation. Typically, installing a new version of a metamodel can result in models that no longer conform to their metamodel and cannot be used with the modelling framework. Moreover, a user discovers conformance problems only when attempting to use a model after installation has completed, and not as part of the installation process.

To enable conformance checking during metamodel installation in EMF, the metamodel-independent syntax has been integrated with a model indexing service, Concordance [Rose *et al.* 2010c]. The work was conducted outside of the scope of the thesis, and is now summarised to indicate the usefulness of the metamodel-independent syntax for supporting the automation of co-evolution activities. Concordance provides a mechanism for resolving inter-model references (such as those between models and their metamodels). Without Concordance, determining the the instances of a metamodel is possible only by checking every model in the workspace. Integrating Concordance and the metamodel-independent syntax resulted in a service, which Epsilon (Section 2.3.2) executes after the installation of a metamodel, to identify the models that are affected by the metamodel changes. All models that conform to the old version of the metamodel are checked for conformance with the new metamodel. As such, conformance checking occurs automatically and immediately after metamodel installation. Conformance problems are detected and reported immediately, rather than when an affected model is next used. Conformance problems detected via Concordance can be reconciled with the structures described in Sections 5.2 and 5.4.

### Summary

Modelling frameworks implicitly enforce conformance, which presents challenges for managing co-evolution. In particular, detecting and reconciling conformance problems involves managing non-conformant models, which cannot be loaded by modelling frameworks and hence cannot be used with model editors or model management operations. The metamodel-independent syntax proposed in this section enables modelling frameworks to load non-conformant models, and has has been integrated with Concordance [Rose *et al.* 2010c] to facilitate the reporting of conformance problems during metamodel installation. The metamodel-independent syntax, binding service and conformance checking service underpin the implementation of the textual modelling notation presented in the sequel. The benefits and drawbacks of the metamodel-independent syntax in the context of user-driven co-evolution are explored in Chapter 6.

## 5.2 Epsilon HUTN: A Textual Modelling Notation

The analysis of co-evolution examples in Chapter 4 highlighted two ways in which co-evolution is managed. In *developer-driven* co-evolution, migration is specified by the metamodel developer in an executable format; while in *user-driven co-evolution* migration is specified by the metamodel developer in prose or not at all. Performing user-driven co-evolution with modelling frameworks presents two key challenges that have not been explored by existing research. Firstly, user-driven co-evolution often involves editing the storage representation of the model, such as XMI. Model storage representations are typically not optimised for human use and hence user-driven co-evolution can be error-prone. Secondly, non-conformant model elements must be identified during user-driven co-evolution. When a multi-pass parser is used to load models, as is the case with EMF, not all conformance problems are reported at once, and user-driven co-evolution is an iterative process. In Section 4.3, these challenges led to the identification of the following requirement: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a sound and complete conformance report for the original model and evolved metamodel.*

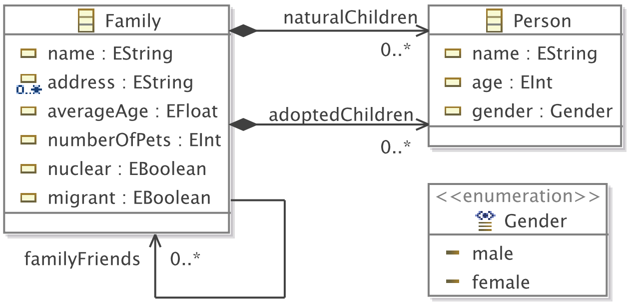
Section 5.2.1 illustrates some of the challenges to performing model migration with XMI, and Section 5.2.2 discusses potential alternatives to XMI for user-driven co-evolution. Section 5.2.3 describes an OMG standard modelling notation that is optimised for human-usability, and Section 5.2.4 presents a reference implementation of the OMG standard that is interoperable with EMF. Finally, Section 5.2.5 demonstrates the way in which the reference implementation of the notation has been integrated with the metamodel independent syntax described in Section 5.1 to produce conformance reports. The work presented in this section has been published in [Rose *et al.* 2008a].

### 5.2.1 Model Migration with XMI

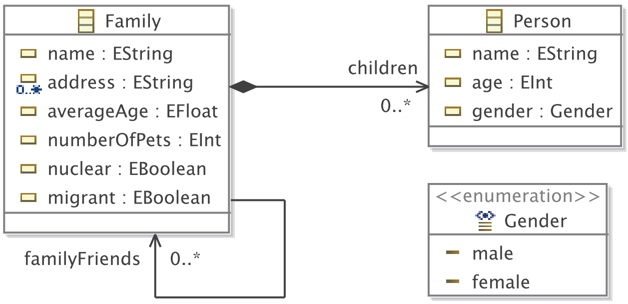
The co-evolution example from Section 5.1 is re-used in this section to illustrate the way in which model migration is performed by editing the underlying storage representation of a model, such as XMI (Section 2.1.3). For convenience, Figures 5.3 and 5.4 are repeated here as Figures 5.13 and 5.14, respectively. Recall that the model (Figure 5.14) conforms to the original metamodel (Figure 5.13(a)).

The model in Figure 5.14 does not conform to the evolved metamodel (because it uses the `naturalChildren` and `adoptedChildren` features, which are not defined for `Person`), and hence cannot be loaded by the modelling framework. Migration might be achieved by editing the underlying storage representation directly (i.e. manually manipulating XMI). Listing 5.3 shows the XMI for the model in Figure 5.14.

```
<?xml version="1.0" encoding="ASCII"?>
<families:Family xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:families="families" xmi:id="_kE2LkAagEeC-FIOYrvUj0A" name="Smiths">
 <naturalChildren xmi:id="_q8RWYAagEeC-FIOYrvUj0A" name="Paul"/>
 <adoptedChildren xmi:id="_nj6TcAagEeC-FIOYrvUj0A" name="John"/>
```

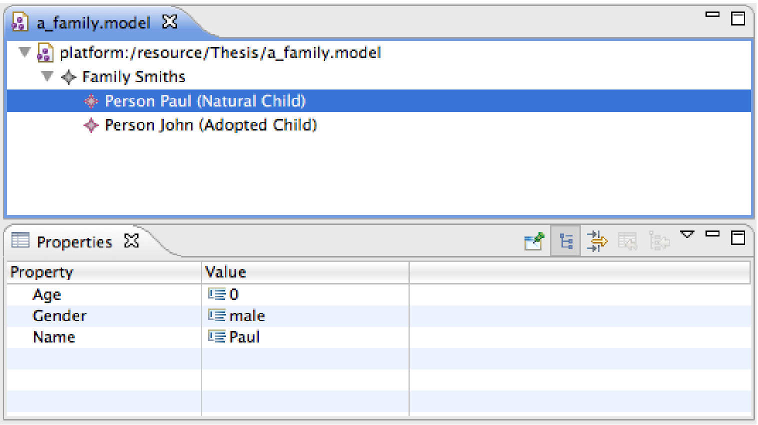


(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.13: Reproduction of Figure 5.3: Evolution of a families metamodel, based on the metamodel in [OMG 2004].



- platform:/resource/Thesis/a_family.model
  - Family Smiths
    - Person Paul (Natural Child)
    - Person John (Adopted Child)

| Property | Value |
|---|---|
| Age | 0 |
| Gender | male |
| Name | Paul |

Figure 5.14: Reproduction of Figure 5.4: A family model, which conforms to the metamodel in Figure 5.13(a)

```
5 </families:Family>
```

Listing 5.3: XMI for the family model in Figure 5.14

XMI is a concrete syntax for models, which has been optimised for use by machines and not by humans [OMG 2004]. Models often contain information that is not relevant to the domain, such as the universally unique identifiers (`xmi:id` attributes) on lines 2, 3 and 4 of Listing 5.3. Furthermore, information is often omitted to reduce the size of the model on disk. For example, the model elements on lines 3 and 4 of Listing 5.3 do not specify their type (`Person`) and this is inferred from the type of the `naturalChildren` and `adoptedChildren` features. Types are inferred from the metamodel by the modelling framework using a reference from the model to the metamodel. In XMI, metamodel references are expressed using XML namespaces. The XMI in Listing 5.3 imports the families metamodel to a namespace (families) on line 2. The evaluation presented in Section 6.1 further explores the suitability of XMI for user-driven co-evolution. The remainder of this section discusses the design and implementation of a modelling notation that provides an alternative to XMI, and is tailored for human-use.

### 5.2.2 Potential Alternatives to XMI

Two characteristics were considered when designing a notation that provides an alternative to representing models with XMI. Models can be represented textually or graphically (Section 2.1.2), and with a metamodel-specific or a metamodel-independent syntax (Section 5.1). The benefits and drawbacks of each option have been considered particularly with respect to their implications for user-driven co-evolution, and are now discussed.

**Metamodel-independent vs metamodel-specific** A metamodel-specific syntax is defined using terms from the metamodel, and is often more concise than a metamodel-independent syntax. A metamodel-specific (and textual) syntax for part of the original families metamodel (Figure 5.13(a)) is shown in Listing 5.4. Using the metamodel-specific syntax, the families model in Listing 5.3 is written `Smiths:Paul(John)`. Notice that the syntax is defined in metamodel terms, such as `Family`, `naturalChildren`, and `adoptedChildren`. Consequently, the syntax definition can be affected by metamodel evolution, and hence cannot be used to load a model that does not conform to the metamodel. Therefore, a metamodel-specific representation is not suitable for use during user-driven co-evolution, which involves using a modelling notation with non-conformant models. Here, a metamodel-independent representation is preferred.

```
family = name ":" naturalChildren "(" adoptedChildren ")"
naturalChildren = name { "," naturalChildren }
adoptedChildren = name { "," adoptedChildren }
name = "A" | ... | "z"
```

Listing 5.4: A metamodel-specific syntax for families in EBNF

**Textual vs graphical** For user-driven co-evolution, the usability of the modelling notation is important because a metamodel user manipulates models with the notation to perform migration. The choice between a textual or graphical notation is likely to have a significant impact on usability, but it was not feasible to conduct a thorough user analysis given the time constraints of the thesis. Instead, a textual notation was selected to reduce implementation effort, and implemented to facilitate the addition of an equivalent graphical notation in future work. In particular, the concrete and abstract syntax definitions of the notation were kept separate to simplify future addition of alternative concrete syntax, such as a graphical notation.

Currently, several tools exist for representing models with textual and metamodel-specific syntaxes (such as the text-to-model transformation tools discussed in Section 2.1.4), but only XMI represents models in a metamodel-independent syntax. The Distributed Systems Technology Centre's TokTok project provided a metamodel-independent textual modelling notation [Steel & Raymond 2001], but the tool has been abandoned and the source code has vanished. Sintaks[2], a tool for constructing metamodel-specific representations, was originally intended to provide a metamodel-independent representation [Muller & Hassenforder 2005].

The TokTok project and earlier versions of Sintaks based their metamodel-independent representations on an OMG standard, Human-Usable Textual Notation (HUTN) [Steel & Raymond 2001, Muller & Hassenforder 2005]. HUTN defines a textual modelling notation that aims to conform to human-usability criteria [OMG 2004]. As a metamodel-independent and textual concrete syntax, HUTN was seen as an ideal starting point for designing a textual modelling notation for user-driven co-evolution. OMG HUTN is described in the sequel.

### 5.2.3 OMG Human-Usable Textual Notation

OMG HUTN is a textual, metamodel-independent modelling notation whose primary design goal is human-usability and "this is achieved through consideration of the successes and failures of common programming languages" [OMG 2004, Section 2.2]. The HUTN specification refers to two studies of programming language usability to justify design decisions. However, the OMG specification cannot evaluate the human-usability of the notation because no reference implementation exists. As HUTN is (purportedly) optimised for human-usability, using HUTN rather than XMI for user-driven co-evolution should lead to increased developer productivity. This claim is explored in Chapter 6.

Like the metamodel presented in Section 5.1, HUTN provides a metamodel-independent syntax for MOF. However, the OMG HUTN specification focuses on concrete syntax, whereas the metamodel-independent syntax presented in Section 5.1 focuses on abstract syntax. In this section, the key features of HUTN are introduced, and the sequel describes a reference implementation of HUTN. Throughout the remainder of this section, the original families metamodel (Figure 5.3(a), 5.13(a)) is used to illustrate

---

[2] `http://www.kermeta.org/sintaks/`

the notation.

### Basic Notation

Listing 5.5 shows the construction of an *object* (an instance of a metamodel class) in OMG HUTN, here an instance of the Family class from Figure 5.13(a). Line 1 specifies the metamodel *package* containing the metamodel classes that can be instantiated by this model (`FamilyPackage`). A package declaration in OMG HUTN is equivalent to a namespace import at the start of an XMI document (e.g. line 2 of Listing 5.3). In Listing 5.5, line 2 names the metamodel class to be instantiated (`Family`) and gives an identifier for the object (`The Smiths`). Package and object identifiers are optional in OMG HUTN, have no direct equivalent in XMI, and are used to specify reference values (discussed below). Lines 3 to 7 define *attribute values*; in each case, the data value is assigned to the attribute with the specified name. The encoding of the value depends on its type: strings are delimited by any form of quotation mark; multi-valued attributes use comma separators, etc. HUTN accesses type information from the metamodel (Figure 5.13(a) here), which the user references in the package declaration (on line 1).

The metamodel in Figure 5.13(a) has a *simple reference* (`familyFriends`) and two *containment references* (`adoptedChildren`; `naturalChildren`). The OMG HUTN representation embeds a contained object directly in the parent object, as shown in Listing 5.6. A simple reference can be specified using the type and identifier of the referred object, as shown in Listing 5.7. Like attribute values, both styles of reference are preceded by the name of the meta-feature.

```
1  FamilyPackage "families" {
2     Family "The Smiths" {
3        nuclear: true
4        name: "The Smiths"
5        averageAge: 25.7
6        numberOfPets: 2
7        address: "120 Main Street", "37 University Road"
8     }
9  }
```

Listing 5.5: Specifying attributes with HUTN, taken from [Rose *et al.* 2008a]

```
FamilyPackage "families" {
   Family "The Smiths" {
      naturalChildren: Person "John" { name: "John" },
                            Person "Jo" { gender: female }
   }
}
```

Listing 5.6: Specifying a containment reference with HUTN, taken from [Rose *et al.* 2008a]

```
FamilyPackage "families" {
   Family "The Smiths" {
      familyFriends: Family "The Does"
   }
   Family "The Does" {}
}
```

Listing 5.7: Specifying a simple reference with HUTN, taken from [Rose *et al.* 2008a]

### Keywords and Adjectives

In general, a metamodel-independent syntax (such as OMG HUTN) is more verbose than a metamodel-specific concrete syntax. However, OMG HUTN defines optional syntactic shortcuts to make model specifications more concise, and aims to make the syntactic shortcuts intuitive [OMG 2004, pg2-4].

Two of the syntactic shortcuts relate to Boolean-valued attributes and are now discussed; a complete list of syntactic shortcuts is provided in the OMG HUTN specification [OMG 2004]. OMG HUTN permits the use of an attribute name to represent the value `true`, or the attribute name prefixed with a tilde to represent the value `false`). When used in the body of the object, this style of Boolean-valued attribute represents a *keyword*. A keyword used to prefix an object declaration is called an *adjective*. Listing 5.8 shows the use of both an attribute keyword (`~nuclear` on line 6) and adjective (`migrant` on line 2), and states that `The Smiths` are `migrant` and that `The Does` are not `nuclear`.

```
1  FamilyPackage "families" {
2     migrant Family "The Smiths" {}
3
4     Family "The Does" {
5        averageAge: 20.1
6        ~nuclear
7        name: "The Does"
8     }
9  }
```

Listing 5.8: Keywords and adjectives in HUTN, taken from [Rose *et al.* 2008a]

### Alternative Reference Syntax

In addition to the syntax defined in Listings 5.6 and 5.7, OMG HUTN defines two alternative syntactic constructs for specifying the value of references. For example, Listing 5.9 demonstrates the use of a reference block for defining `The Does` as friends with both `The Smiths` and `The Bloggs`. Listing 5.10 illustrates a further alternative syntax for references, which employs an infix notation.

```
FamilyPackage "families" {
   Family "The Smiths" {}
```

```
   Family "The Does" {}
   Family "The Bloggs" {}

   familyFriends {
      "The Does" "The Smiths"
      "The Does" "The Bloggs"
   }
}
```

Listing 5.9: A reference block in HUTN, taken from [Rose *et al.* 2008a]

```
FamilyPackage "families" {
   Family "The Smiths" {}
   Family "The Does" {}
   Family "The Bloggs" {}

   Family "The Smiths" familyFriends Family "The Does";
   Family "The Smiths" familyFriends Family "The Bloggs";
}
```

Listing 5.10: An infix reference in HUTN, taken from [Rose *et al.* 2008a]

The reference block (Listing 5.9) and infix (Listing 5.10) notations are syntactic variations on – and have identical semantics to – the reference notation shown in Listings 5.6 and 5.7.

#### Customisation via Configuration

The OMG HUTN specification allows some limited, metamodel-specific customisation of the notation, using *configuration rules*. Customisations include a parametric form of object instantiation; renaming of metamodel elements; specifying the default value of a feature; and providing a default identifier for classes of object.

### 5.2.4 Reference Implementation: Epsilon HUTN

To investigate the extent to which OMG HUTN can be used for user-driven co-evolution, a reference implementation, Epsilon HUTN, has been designed and implemented. This section describes the way in which Epsilon HUTN was implemented using a combination of model-management operations. From text conforming to the OMG HUTN syntax (described above), Epsilon HUTN produces an equivalent model that can be managed with EMF (Section 2.3.1). The sequel demonstrates the way in which Epsilon HUTN can be used for user-driven co-evolution.

#### Design of Epsilon HUTN

Implementing OMG HUTN involved building a tool for producing an EMF model (i.e. a model represented in XMI) from text conforming to the OMG HUTN syntax (described above). Essentially then, Epsilon HUTN can be regarded as a parser (that emits models), or as a text-to-model transformation. Several approaches to constructing Epsilon HUTN were considered, including: using a text-to-model (T2M) transformation tool (Section 2.1.4), using a domain-specific language (DSL) framework (Section 2.4.1), and using MDE tools and techniques such as EMF (Section 2.3.1), Epsilon (Section 2.3.2) and metamodelling.

As was the case for the design and implementation of the metamodel-independent syntax (Section 5.1), it was preferrable to avoid dependencies on tools that were not part of the Eclipse Modelling Project (in order not to complicate installation of the notation for users). In 2008, the Eclipse Modeling Project[3] did not provide a standard T2M language or DSL framework, and so these implementation strategies were discounted.

Instead, Epsilon HUTN was constructed using existing languages of the Epsilon platform. To parse HUTN source, a parser was generated with the ANTLR parser generator tool [Parr 2007], which had been used successfully to implement parsers for every task-specific language of Epsilon [Kolovos 2009]. A parser generated with ANTLR emits an abstract syntax tree (a set of Java objects that conform to a simple tree data structure), from which the Epsilon HUTN tool needs to produce an EMF model.

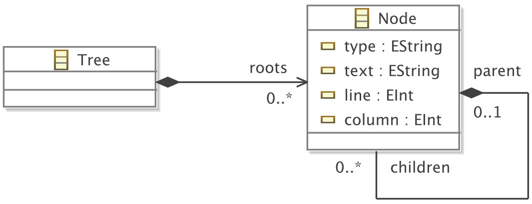


Figure 5.15: A metamodel for abstract syntax trees, in Ecore

The abstract syntax tree produced by ANTLR can be regarded as a model (conforming to the metamodel in Figure 5.15) and hence, producing an EMF model from the abstract syntax tree can be regarded as a model-to-model transformation. Epsilon HUTN, however, was designed as two separate model-to-model transformations, for two reasons. Firstly, initial prototyping highlighted that the difference between a model represented in terms of the tree metamodel in Figure 5.15 and the same model represented in metamodel-specific terms is vast, and the logic required to perform a one-step transformation quickly became complicated even for simple models. In particular, each transformation rule would have required a lengthly guard statement to distinguish `Nodes` representing different types of model element, which would have been difficult to debug and maintain. Secondly, it became apparent that the concrete syntax defined in OMG HUTN could be transformed to the metamodel-independent

---

[3] `http://www.eclipse.org/modeling/`

syntax defined in Section 5.1, which would reduce implementation effort by re-using the metamodel and conformance checking service described in Section 5.1.

### Implementation of Epsilon HUTN

For the reasons outlined above, Epsilon HUTN is implemented using two model-to-model transformations. Figure 5.16 outlines the workflow through Epsilon HUTN, from HUTN source text to an EMF instantiation of the target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [Parr 2007]. From this, a Java postprocessor is used to construct an instance of the simple AST metamodel in Figure 5.15. Using ETL, a M2M transformation is applied to produce an intermediate model, which is an instance of the metamodel-independent syntax discussed in Section 5.1. Validation is performed on the intermediate model to ensure that the syntactic constraints specified in the OMG HUTN specification are satisfied[4], and that the model conforms to the target metamodel. Conformance checking is achieved by re-using the service presented in Section 5.1. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, which consumes the intermediate model and produces the target model[5].

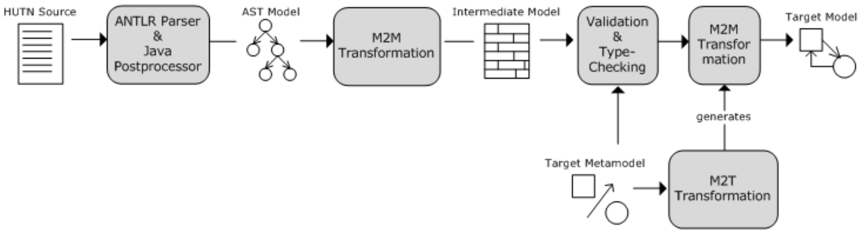


Figure 5.16: The architecture of Epsilon HUTN.

The modular architecture in Figure 5.16 facilitated the re-use of the metamodel-independent syntax and conformance checking service described in Section 5.1, and hence reduced implementation effort. A small modification was made to the metamodel-independent syntax to facilitate the implementation of Epsilon HUTN: an additional metaclass, `PackageObject`, was added to the metamodel-independent syntax. In OMG HUTN, packages are used to segregate a model such that different parts of a OMG HUTN document can refer to different metamodels. Consequently, a `PackageObject` has a type (i.e. the metamodel to which its contents refer), an optional identifier (used for inter-package references) and contains any number of `Objects`. To avoid confusion with `PackageObject`, the `Object` class in the metamodel-independent

---

[4] For example, no two objects may have the same identifier.

[5] This final step involves a higher-order transformation (a M2T transformation is used to produce a M2M transformation), and is described in more detail below.

syntax was renamed to `ClassObject`. The version of the metamodel-independent syntax used with Epsilon HUTN is shown in Figure 5.17.

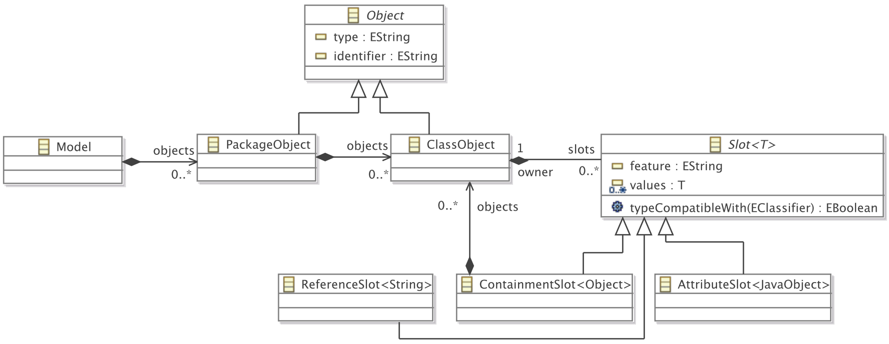


Figure 5.17: Final version of the metamodel-independent syntax, in Ecore

Each module of the architecture in Figure 5.16 is now discussed in detail. Note that, in this section, instances of the metamodel-independent syntax produced during the execution of the HUTN workflow are termed an *intermediate model*.

**Parsing the HUTN Source** A parser for OMG HUTN was constructed using ANTLR [Parr 2007], a parser generator tool. ANTLR produces a parser from an annotated EBNF grammar definition. To simplify the implementation of the model transformations described below, the EBNF grammar used by Epsilon HUTN varies slightly from the grammar defined in OMG HUTN. Part of the grammar definition used by Epsilon HUTN is shown in Listing 5.11 and is used to generate parser rules that process the body of `ClassObjects`. The `attr` rule on line 4, for example, matches any number of comma separated attribute values or the `null` keyword.

Epsilon HUTN uses a simple, bespoke Java post-processor to construct instances of the abstract syntax tree metamodel (Figure 5.15) from the Java objects produced by ANTLR. Specifically, the post-processor copies the Java objects produced by the parser into an EMF resource, and hence produces a model that can be managed with EMF.

```
1  cls_contents = feature | adjective
2  feature = NAME ASSIGNMENT feature_contents
3  feature_contents = attr | refs | containments
4  attr = attr_value { COMMA attr } | NULL
```

Listing 5.11: An extract of the Epsilon HUTN grammar definition in EBNF

**AST Model to Intermediate Model** For M2M transformation, Epsilon HUTN uses ETL [Kolovos *et al.* 2008b]. One of the transformation rules from Epsilon HUTN is shown in Listing 5.12; the complete transformation is presented in Listing A.3. In

Listing 5.12, the rule (starting on line 1) transforms a `Node` with type `name` (which could represent a `PackageObject` or a `ClassObject`) to a `PackageObject` in the intermediate model. The guard (line 5) specifies that a name node will only be transformed to a `PackageObject` if the node has no parent (i.e. it is a top-level node, and hence a package rather than a class). The body of the rule states that the type, line number and column number of the package are determined from the text, line and column attributes of the `Node` object. On line 11, the children of the `Node` object are transformed to the intermediate model (using a method built into ETL, `equivalent()`), and added to the `objects` reference of the `PackageObject`.

```
1  rule NameNode2PackageObject
2     transform n : AntlrAst!Node
3     to p : Intermediate!PackageObject {
4
5     guard : n.type == ’Name’ and n.parent.isUndefined()
6
7     p.type := n.text;
8     p.line := n.line;
9     p.col := n.column;
10
11    p.objects.addAll(n.children.equivalent());
12 }
```

Listing 5.12: Transforming Nodes to PackageObjects with ETL.

**Intermediate Model Validation** An advantage of the two-stage transformation is that contextual analysis can be specified in an abstract manner – that is, without having to express the traversal of the AST. This gives clarity and minimises the amount of code required to define syntactic constraints.

```
context ClassObject {
   constraint IdentifiersMustBeUnique {
      guard: self.id.isDefined()
      check: ClassObject.all
                .select(c|c.id = self.id).size() = 1;
      message: ‘Duplicate identifier: ’ + self.id
   }
}
```

Listing 5.13: A constraint (in EVL) to check that all identifiers are unique

Epsilon HUTN uses EVL [Kolovos *et al.* 2009] to specify validation, resulting in highly expressive syntactic constraints. An EVL constraint comprises a guard, the logic that specifies the constraint, and a message to be displayed if the constraint is not met. For example, Listing 5.13 specifies the constraint that every HUTN class object has a unique identifier. The complete set of constraints is presented in Listing A.4.

In addition to the syntactic constraints defined in the OMG HUTN specification, the EVL constraints for checking conformance (Section 5.1) are executed on the model at this stage.

**Intermediate Model to Target Model** When the intermediate model conforms to the target metamodel, the intermediate model can be transformed to an instance of the target metamodel. In other words, the model can be represented in a metamodel-specific manner and, for example, used with model management operations. In generating the target model from the intermediate model (Figure 5.16), the transformation uses information from the target metamodel, such as the names of classes and features. A typical approach to this category of problem is to use a higher-order transformation (HOT) on the target metamodel to generate the desired transformation [Tisi *et al.* 2009]. Currently, ETL cannot be used to produce a transformation from a transformation and hence Epsilon HUTN uses a slightly different approach: the transformation to the target model is produced by executing a M2T transformation on the target metamodel, using EGL [Rose *et al.* 2008b]. EGL is a template-based M2T language; `[% %]` tag pairs are used to denote dynamic sections, which may produce text when executed; any code not enclosed in a `[% %]` tag pair is included verbatim in the generated text.

Listing 5.14 shows part of the M2T transformation used by Epsilon HUTN; the complete M2T transformation is presented in Listing A.5. When executed on the target metamodel, the M2T transformation generates an ETL program (i.e. a M2M transformation). The generated ETL code transforms an intermediate model to a model that conforms to the target metamodel. The loop beginning on line 1 of Listing 5.14 iterates over each metaclass in the target metamodel, producing a M2M transformation rule. The generated transformation rule consumes a `ClassObject` in the intermediate model and produces an element of the target model. The guard of the generated transformation rule (line 6) ensures that only `ClassObject` with a type equal to the current meta-class are transformed by the generated rule. To generate the body of the rule, the M2T transformation iterates over each structural feature of the current meta-class, and generates appropriate transformation code for populating the values of each structural feature from the slots on the class object in the intermediate model. The part of the M2T transformation that generates the body of the M2M transformation rule is omitted in Listing 5.14 because it contains a large amount of code for interacting with EMF, which is not relevant to this discussion. The complete M2T transformation is presented in Listing A.5.

```
1  [% for (class in EClass.allInstances()) { %]
2  rule Object2[%=class.name%]
3    transform o : Intermediate!ClassObject
4    to t : Model![%=class.name%] {
5
6      guard: o.type = ‘[%=class.name%]’
7
8      -- body omitted
```

```
  }
[% } %]
```

Listing 5.14: Part of the M2T transformation (in EGL) that takes a target metamodel and generates an intermediate model to target model transformation (in ETL).

To illustrate the way in which Epsilon HUTN generates a target model from an intermediate model, the M2T transformation in Listing 5.14 is applied to the the families metamodel in Figure 5.13(a). The M2T transformation generates the two M2M transformation rules in Listing 5.15. The rules produce instances of `Family` and `Person` from instances of `ClassObject` in the intermediate model. The body of each rule copies the values from the slots of the `ClassObject` to the `Family` or `Person` in the target model. Lines 7-9, for example, copy the value of the name `Slot` (if one is specified) to the target `Family`.

Currently, Epsilon HUTN can be used only to generate EMF models. Support for other modelling languages would require different transformations between intermediate and target model. In other words, for each target modelling language, a new EGL template would be required. The transformation from AST to intermediate model is independent of the target modelling language and would not need to change. As EMF is arguably the most widely-used modelling framework today, support for other modelling frameworks is not crucial for exploring the suitability of HUTN for user-driven co-evolution. However, one interesting example of metamodel evolution predates EMF: the changes made to UML between versions 1.5 and 2.0 of the specification. Because the UML 1.x specifications use a version of MOF that is not supported by EMF, the current version of Epsilon HUTN cannot be used for migrating UML 1.x models. Several other examples, however, were available for evaluating Epsilon HUTN, and so support for other modelling frameworks was not crucial in the context of the thesis research.

### Compliance with OMG HUTN

Epsilon HUTN is a reference implementation of the OMG HUTN standard. There are, however, a few differences between the implementation in Epsilon and the OMG standard. The differences are now discussed and justified. The discussion is based on the Epsilon HUTN compliance report[6], which provides up-to-date information on compliance with the OMG HUTN standard.

Table 5.2 summarises the differences between Epsilon HUTN and the OMG HUTN standard. Epsilon HUTN does not support two of the syntactic shortcuts described for classes in the OMG HUTN standard: parametric attributes and enumeration adjectives. The former are used to specify attribute values in a parametric form (e.g. `Point (0,0)`, for creating a `Point` object with `x` and `y` attributes with value 0). The latter allows an enumeration value to prefix an object definition (e.g. `female Person` for creating a `Person` with `female` gender). The attribute to which the parametric

---

[6] `http://www.eclipse.org/gmt/epsilon/doc/articles/hutn-compliance/`

```
rule Object2Family
 transform o : Intermediate!ClassObject
 to t : Model!Family {

  guard: o.type = 'Family'

  if (o.hasSlot('name')) {
   t.name := o.findSlot('name').values.first;
  }

  if (o.hasSlot('address')) {
   for (value in o.findSlot('address').values) {
    t.address.add(value);
   }
  }

  -- remainder of body omitted
 }

rule Object2Person
 transform o : Intermediate!ClassObject
 to t : Model!Person {

  guard: o.type = 'Person'

  if (o.hasSlot('name')) {
   t.name := o.findSlot('name').values.first;
  }

  -- remainder of body omitted
 }
```

Listing 5.15: The M2M transformation generated for the Families metamodel

or enumeration values are bound is specified using OMG HUTN configuration rules (Section 5.2.3). Parametric attribute and enumeration adjectives were not implemented to reduce the amount of time required to build Epsilon HUTN, but could be easily added in the future. No functionality is lost through these omissions, as alternative (albeit less concise) notion can be used to express models without using parametric attribute and enumeration adjectives.

Section 6.4 of the OMG HUTN standard [OMG 2004] appears to contain a mistake in the grammar definition. Grammar rule 20 implies that an attribute's name is optional when specifying a keyword attribute, and that an empty string and a tilde character are valid forms of a keyword attribute. However, the prose describing grammar rule 20 proposes no semantics for an empty string or a tilde character as a keyword attribute.

| **OMG HUTN** | **Epsilon HUTN** | |
|---|---|---|
| **Feature** | **Supported?** | **Details of support** |
| Packages | Yes | |
| Classes | Partial | Not yet supported: parametric attributes, enumeration adjectives. |
| Attributes | Yes | Corrects a mistake in the standard. |
| References | Yes | |
| Classifier-level attributes | Yes | |
| Data values | Yes | |
| Inline configuration | No | A configuration model is used instead. |
| Configuration rules | Partial | Not yet supported: parametric attributes, enumeration adjectives. |

Table 5.2: Compliance of Epsilon HUTN with OMG HUTN

Consequently, Epsilon HUTN deviates from grammar rule 20 of the OMG HUTN standard, and requires an attribute name for every keyword attribute.

Finally, the OMG HUTN standard defines syntax for specifying configuration rules *inline*, at the start of a HUTN document. Epsilon HUTN does not support inline configuration, and Epsilon HUTN documents are configured with a configuration model, which is constructed using an EMF model editor. Using a configuration model rather than inline configuration reduced the time required to implement Epsilon HUTN and facilitated re-use of configuration models between HUTN documents.

The OMG HUTN standard does not include a set of compliance tests for reference implementations. Instead, the compliance of Epsilon HUTN to OMG HUTN was checked using the many examples of HUTN documents in the OMG HUTN standard [OMG 2004]. The examples were used to create a suite of executable compliance test cases, which were run frequently during the development of Epsilon HUTN.

### 5.2.5 Migration with Epsilon HUTN

Used in combination with the metamodel-independent syntax presented in Section 5.1, Epsilon HUTN facilitates user-driven co-evolution using the workflow in Section 5.18, which provides an alternative to the user-driven co-evolution workflow observed in Section 4.2.2. First, the user attempts to load a model with the model editor[7]. If the model is non-conformant and cannot be loaded, the user clicks a “Generate HUTN” menu item provided by Epsilon HUTN. Epsilon HUTN then binds the model to the metamodel-

---

[7] The workflow in Figure 5.18 assumes a graphical model editor, such as those generated by GMF, but any editor built with EMF will exhibit the same behaviour.

```
FamilyPackage "families" {
   Family "Smiths" {
      name: "Smiths"
      naturalChildren: Person { name: "Paul" }
      adoptedChildren: Person { name: "John" }
   }
}
```

Listing 5.16: OMG HUTN for people with mothers and fathers.

independent syntax and unparses the bound model to produce HUTN source code equivalent to XMI representation of the non-conformant model.

To support the final step of the workflow in Figure 5.18, Epsilon HUTN provides an editor for HUTN documents that is integrated with the conformance checking service (described in Section 5.1). The user edits the HUTN document to reconcile conformance problems (i.e. performs migration), and Epsilon HUTN automatically performs conformance checking as the user edits the HUTN document. When the conformance problems are fixed, the user saves the HUTN document and Epsilon HUTN automatically generates XMI for the conformant model (using the model transformations described in Section 5.2.4). The conformant model can then be loaded with the model editor.

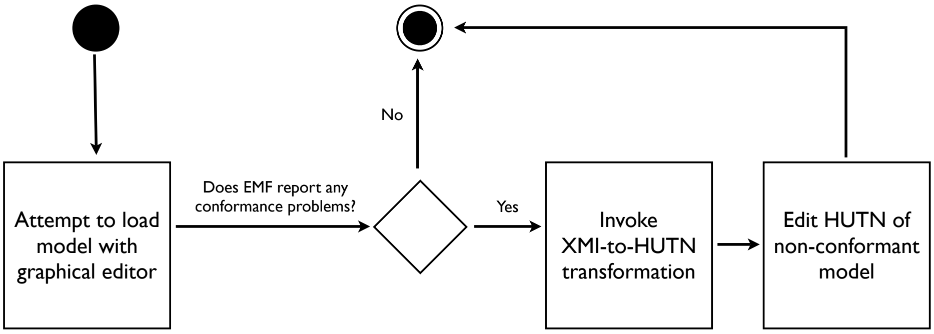
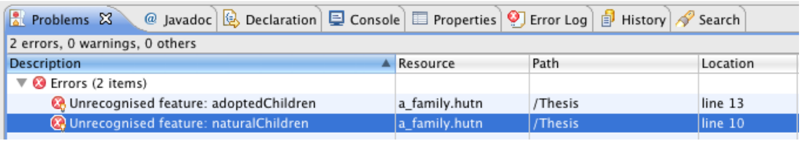


Figure 5.18: User-driven co-evolution with dedicated structures

To demonstrate the way in which HUTN can be used to perform migration, the XMI shown in Listing 5.3 is represented using OMG HUTN in Listing 5.16. Recall that the XMI describes a `Family` with one adopted and one natural child.

If the Families metamodel now evolves such that children are modelled using one rather than two features (Figure 5.13(b)), Epsilon HUTN reports conformance problems on the HUTN document using the conformance checking service described in Section 5.1, as illustrated by the screenshot in Figure 5.19.

Resolving the conformance problems requires the user to merge the values for `adoptedChildren` and `naturalChildren` into a set of values for the new feature,



2 errors, 0 warnings, 0 others

| Description | Resource | Path | Location |
|---|---|---|---|
| Errors (2 items) | | | |
| Unrecognised feature: adoptedChildren | a_family.hutn | /Thesis | line 13 |
| Unrecognised feature: naturalChildren | a_family.hutn | /Thesis | line 10 |

Figure 5.19: Conformance problem reporting in Epsilon HUTN.

```
FamilyPackage "families" {
   Family "Smiths" {
      name: "Smiths"
      children: Person { name: "Paul" },
                Person { name: "John" }
   }
}
```

Listing 5.17: HUTN for people with parents.

`children`. The Epsilon HUTN development tools provide content assistance, which might be useful in this situation. Listing 5.17 shows a HUTN document that conforms to the evolved metamodel in which adopted and natural children are specified using a single feature, `children`.

When the user saves the reconciled HUTN document, Epsilon HUTN will automatically generate XMI for the (now) conformant model, and migration is complete. Compared to the user-driven co-evolution workflow observed in Section 4.2.2, the workflow presented in Figure 5.18 provides live conformance checking and a modelling notation that is optimised for humans rather than for machines. The two workflows are compared and evaluated in Chapter 6.

### 5.2.6 Summary

In this section, a textual modelling notation for performing model migration has been designed and implemented. The notation proposed in this section is based on the OMG HUTN standard, which was described in Section 5.2.3. The design and implementation of Epsilon HUTN, an implementation of OMG HUTN for EMF, was discussed in this section. Integration of Epsilon HUTN with the metamodel-independent syntax in Section 5.1 facilitates user-driven co-evolution with a textual modelling notation other than XMI, as demonstrated by the example above. The user-driven co-evolution workflow presented in Section 5.2.5 is evaluated in Chapter 6. The remainder of this chapter focuses on developer-driven co-evolution, in which model migration strategies are executable.

## 5.3 An Analysis of Languages used for Model Migration

In contrast to the previous section, this section focuses on *developer-driven* co-evolution, in which migration is specified as a program that metamodel users execute to migrate their models. Section 4.2.3 discussed existing approaches to model migration, highlighting variation in the languages used for specifying migration strategies. In this section, the results of comparing migration strategy languages are described, using a new example of metamodel evolution (Section 5.3.1). From the comparison, requirements for a domain-specific language for specifying and executing model migration strategies are derived (Section 5.3.3). The sequel describes an implementation of a model migration language based on the analysis presented here. The work described in this section has been published in [Rose *et al.* 2010f].

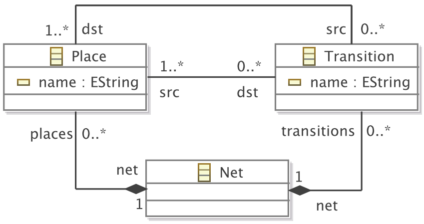
### 5.3.1 Co-Evolution Example

The Petri net metamodel evolution is now used to compare model migration languages. The example has been used often in co-evolution literature, and hence is a useful benchmark for migration languages [Cicchetti *et al.* 2008, Garcés *et al.* 2009, Wachsmuth 2007].

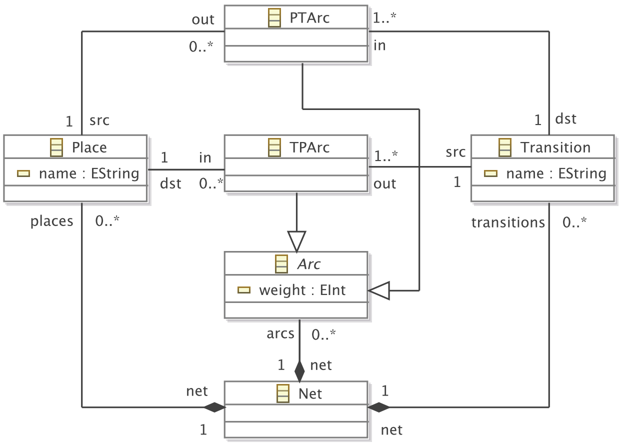
In Figure 5.20(a), a Petri `Net` is defined to comprise `Places` and `Transitions`. A `Place` has any number of `src` or `dst` `Transitions`. Similarly, a `Transition` has at least one `src` and `dst` `Place`. The metamodel is to be evolved to support weighted connections between `Places` and `Transitions` and between `Transitions` and `Places`, as shown in Figure 5.20(b). `Places` are connected to `Transitions` via instances of `PTArc`. Likewise, `Transitions` are connected to `Places` via `TPArc`. Both `PTArc` and `TPArc` inherit from `Arc`, and can be used to specify a `weight`.

Models that conform to the original metamodel might not conform to the evolved metamodel. The following strategy can be used to migrate models:

```
1. For every instance, t, of Transition:
       For every Place, s, referenced by the src feature of t:
           Create a new instance, arc, of PTArc.
           Set s as the src of arc.
           Set t as the dst of arc.
           Add arc to the arcs reference of the Net referenced by t.
       For every Place, d, referenced by the dst feature of t:
           Create a new instance, arc, of TPArc.
           Set t as the src of arc.
           Set d as the dst of arc.
           Add arc to the arcs reference of the Net referenced by t.
2. And nothing else changes.
```



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.20: Petri nets metamodel evolution. Taken from [Rose *et al.* 2010f].

### 5.3.2 Languages Currently Used for Model Migration

Using the above example, the languages used by existing approaches for specifying and executing model migration strategies are now compared. From this comparison, the strengths and weakness of each language are highlighted and requirements for a model migration language are synthesised in the sequel.

#### Manual Specification with M2M Transformation

Model migration can be specified using M2M transformation. For example, the Petri net migration has been specified in the M2M transformation language, ATL (Atlas Transformation Language) [Jouault & Kurtev 2005], with an inference co-evolution approach [Garcés *et al.* 2009]. This is reproduced in Listing 5.18. Rules for migrating `Places` and `TPArcs` have been omitted for brevity, but are similar to the `Nets` and `PTArcs` rules.

Model transformation in ATL is specified using `rules`, which transform source

```
 1  rule Nets {
 2    from o : Before!Net
 3    to m : After!Net ( places <- o.places, transitions <- o.transitions )
 4  }
 5
 6  rule Transitions {
 7    from o : Before!Transition
 8    to m : After!Transition (
 9       name <- o.name,
10       "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
11       out <- o.dst->collect(p | thisModule.TPArcs(o,p))
12     )
13  }
14
15  unique lazy rule PTArcs {
16    from place : Before!Place, destination : Before!Transition
17    to ptarcs : After!PTArc (
18       src <- place, dst <- destination, net <- destination.net
19     )
20  }
```

Listing 5.18: Part of the Petri nets migration in ATL, from [Rose *et al.* 2010f]

model elements (specified using the `from` keyword) to target model elements (specified using `to` keyword). For example, the `Nets` rule on line 1 of Listing 5.18 transforms an instance of `Net` from the original (source) model to an instance of `Net` in the evolved (target) model. The source model element (the variable `o` in the `Net` rule) is used to populate the target model element (the variable `m`). ATL allows rules to be specified as *lazy* (not scheduled automatically and applied only when called by other rules).

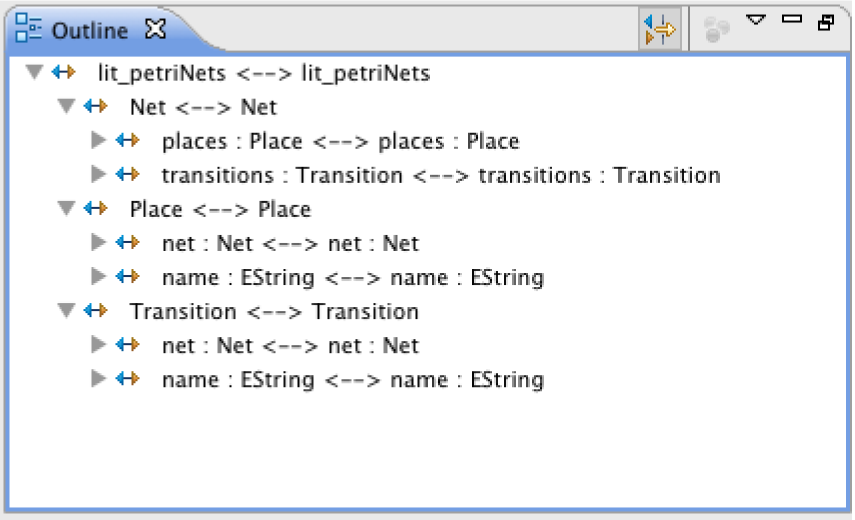
The `Transitions` rule in Listing 5.18 codifies in ATL the migration strategy described previously. The rule is executed for each `Transition` in the original model, `o`, and constructs a `PTArc` (`TPArc`) for each reference to a `Place` in `o.src` (`o.dst`). Lazy rules must be used to produce the arcs to prevent circular dependencies with the `Transitions` and `Places` rules. Here, ATL, a typical rule-based transformation language, is considered and model migration would be similar in QVT. With Kermeta, migration would be specified in an imperative style using statements for copying `Nets`, `Places` and `Transitions`, and for creating `PTArcs` and `TPArcs`.

In model transformation, two common categories of relationship between source and target model, *new-target* and *existing-target*, have been identified (Section 2.1.4). In the former, the target model is constructed afresh by the execution of the transformation, while in the latter, the target model contains the same data as the source model before the transformation is executed. M2M transformation languages typically support new-target transformations. Some M2M transformation languages also support existing-target transformations, but typically require the source and target metamodel to be identical.

In model migration, source and target metamodels differ, and therefore existing-target transformations cannot be used to specify model migration strategies. Consequently, model migration strategies are specified with new-target model-to-model transformation languages, and often contain sections for copying from original to migrated model those model elements that have not been affected by metamodel evolution. For the Petri nets example, the `Nets` rule (in Listing 5.18) and the `Places` rule (not shown) exist only for this reason.

### Manual Specification with a Metamodel Mapping

Model migration can be undertaken using the model loading mechanisms of EMF [Hussey & Paternostro 2006], with a tool that is termed *Ecore2Ecore* here. EMF binds models to a specific metamodel, and hence cannot be used to load models that have been affected by metamodel evolution (Section 4.2.1). Therefore, Ecore2Ecore requires the metamodel developer to provide a mapping between the metamodelling language of EMF (Ecore) and the concrete syntax used to persist models (XMI). Mappings are specified using Ecore2Ecore, which can suggest relationships between source and target metamodel elements by comparing names and types. Figure 5.21 shows mappings between the original and evolved Petri nets metamodels.



- lit_petriNets <--> lit_petriNets
  - Net <--> Net
    - places : Place <--> places : Place
    - transitions : Transition <--> transitions : Transition
  - Place <--> Place
    - net : Net <--> net : Net
    - name : EString <--> name : EString
  - Transition <--> Transition
    - net : Net <--> net : Net
    - name : EString <--> name : EString

Figure 5.21: Mappings between the original and evolved Petri nets metamodels, constructed with the tool described in [Hussey & Paternostro 2006]

The mappings are used by the EMF XMI parser to determine the metamodel types to which pieces of the XMI will be bound. When a type or feature is not bound, the user must specify a custom migration strategy in Java. For the Petri nets metamodel, the `src` and `dst` features of `Place` and `Transition` are not bound, because migration is more complicated than a one-to-one mapping.

```
private Collection<Place> toCollectionOfPlaces
(String value, Resource resource) {

  final String[] uriFragments = value.split("␣");
  final Collection<Place> places = new LinkedList<Place>();

  for (String uriFragment : uriFragments) {
    final EObject eObject = resource.getEObject(uriFragment);
    final EClass place = PetriNetsPackage.eINSTANCE.getPlace();

    if (eObject == null || !place.isInstance(eObject))
      // throw an exception

    places.add((Place)eObject);
  }

  return places;
}
```

Listing 5.19: Java method for deserialising a reference.

In Ecore2Ecore, model migration is specified on the XMI representation of the model and requires some knowledge of the XMI standard. For example, in XMI, references to other model elements are serialised as a space delimited collection of URI fragments [Steinberg *et al.* 2008]. Listing 5.19 shows a fragment of the code used to migrate Petri net models with Ecore2Ecore. The method shown converts a `String` containing URI fragments to a `Collection` of `Places`. The method is used to access the `src` and `dst` features of `Transition`, which no longer exist in the evolved metamodel and hence are not loaded automatically by EMF. To specify the migration strategy for the Petri nets example, the metamodel developer must know the way in which the `src` and `dst` features are represented in XMI. The complete listing (presented in Section A.1) exceeds 150 lines of code.

### Operator-based Co-evolution with COPE

Operator-based approaches to identifying and managing co-evolution, such as COPE [Herrmannsdoerfer *et al.* 2009a], provide a library of *co-evolutionary operators*. Each co-evolutionary operator specifies both a metamodel evolution and a corresponding model migration strategy. For example, the "Make Reference Containment" operator from COPE evolves the metamodel such that a non-containment reference becomes a containment reference and migrates models such that the values of the evolved reference are replaced by copies. By composing co-evolutionary operators, metamodel evolution can be performed and a migration strategy can be generated without writing any code.

Clearly, the development of an operator-based approach must start by identifying

```
for (transition in petrinets.Transition.allInstances) {
  for (source in transition.unset(’src’)) {
    def arc = petrinets.PTArc.newInstance()
    arc.src = source
    arc.dst = transition
    arc.net = transition.net
  }

  for (destination in transition.unset(’dst’)) {
    def arc = petrinets.TPArc.newInstance()
    arc.src = transition
    arc.dst = destination
    arc.net = transition.net
  }
}

for (place in petrinets.Place.allInstances) {
  place.unset(’src’)
  place.unset(’dst’)
}
```

Listing 5.20: Petri nets model migration in COPE

operators, which can be challenging. Operators capture both metamodel evolution and model migration semantics, and as such, a complete library of operators is difficult to imagine [Lerner 2000]. Instead, operator-based approaches seek to capture the most commonly occurring co-evolutionary operators, which are typically identified by examining existing examples of evolution [Herrmannsdoerfer *et al.* 2008a]. Hence, the breadth of consultation when identifying operators will affect the efficacy of an operator-based approach, because evolutionary changes that occur frequently in one project might never occur in other projects, or vice-versa.

To perform metamodel evolution using an operator-based approach, the library of co-evolutionary operators must be integrated with tools for editing metamodels. COPE provides integration with the EMF tree-based metamodel editor. Operators may be applied to an EMF metamodel, and COPE tracks their application. Once metamodel evolution is complete, a migration strategy can be generated automatically from the record of changes maintained by COPEs. The migration strategy is distributed along with the updated metamodel, and metamodel users choose when to execute the migration strategy on their models.

To be effective, operator-based approaches must provide a rich yet navigable library of co-evolutionary operators (Section 4.2.3). COPE allows model migration strategies to be specified manually when no co-evolutionary operator is appropriate. COPE employs a fundamentally different approach to M2M transformation and Ecore2Ecore, using an existing-target transformation. As discussed above, existing-target transformations cannot be used for specifying model migration strategies as the source (original) and target (evolved) metamodels differ. However, models can be structured independently of their metamodel using a metamodel-independent syntax (such as the one introduced in Section 5.1).

Listing 5.20 shows the COPE model migration strategy for the Petri net example given above[8]. Most notably, slots for features that no longer exist must be explicitly `unset`. In Listing 5.20, slots are `unset` on four occasions (on lines 2, 9, 18 and 19), once for each feature that is in the original metamodel but not in the evolved metamodel. These features are: `src` and `dst` of `Transition` and of `Place`. Failing to unset slots that do not conform to the evolved metamodel causes migration to fail with an error.

### 5.3.3 Requirements Identification

Requirements for a domain-specific language for model migration were identified from the review of existing languages (Section 5.3.2). The derivation of the requirements is now summarised, by considering two orthogonal concerns: the source-target relationship of the language used for specifying migration strategies and the way in which models are represented during migration.

#### Source-Target Relationship Requirements

When migration is specified as a new-target transformation, as in ATL (Listing 5.18), model elements that have not been affected by metamodel evolution must be explicitly copied from the original to the migrated model. When migration is specified as an existing-target transformation, as in COPE (Listing 5.20), model elements and values that no longer conform to the target metamodel must be explicitly removed from the migrated model. Ecore2Ecore does not require explicit copying or unsetting code; instead, the relationship between original and evolved metamodel elements is captured in a mapping model specified by the metamodel developer. The mapping model can be derived automatically and customised by the metamodel developer. To explore the appropriateness for model migration of an alternative to new- and existing-target transformations, the following requirement was derived:

*The migration language must* ***automatically*** *copy every model element that conforms to the evolved metamodel from original to migrated model, and must automatically not copy any model element that does not conform to the evolved metamodel from original to migrated model.*

#### Model Representation Requirements

With Ecore2Ecore, migration is achieved by manipulating XMI. Consequently, the metamodel developer must be familiar with XMI and must perform tasks such as

---

[8] In Listing 5.20, some of the concrete syntax has been changed in the interest of readability.

dereferencing URI fragments (Listing 5.19) and type conversion. Transformation languages abstract over the underlying storage representation of models (such as XMI) by using a modelling framework to load, store and access models.

*The migration language must not expose the underlying representation of models.*

To apply co-evolution operators, COPE requires the metamodel developer to use a specialised metamodel editor. The editor can manipulate only metamodels defined with EMF. Similarly, the mapping tool used in the Ecore2Ecore approach can be used only with metamodels defined with EMF. Although EMF is arguably very widely-used, other modelling frameworks exist. Adapting to interoperate with new systems is recognised as a common reason for software evolution [Sjøberg 1993], and migration between modelling frameworks is as a possible use case for a model migration language. In particular, there is demand for migrating between UML 1 (e.g. [OMG 2001]) and UML 2 (e.g. [OMG 2007b]) models[9], which are typically managed with different modelling frameworks. Decoupling model management operations from the model representation facilitates interoperability with many modelling technologies, as demonstrated by Epsilon (Section 2.3.2). Therefore, to facilitate interoperability with modelling frameworks other than EMF, the following requirement was derived:

*The migration language must be loosely coupled with modelling frameworks and must not assume that models and metamodels will be represented in EMF.*

## 5.4 Epsilon Flock: A Model Migration Language

Driven by the analysis presented above, a domain-specific language for model migration, Epsilon Flock (subsequently referred to as Flock), has been designed and implemented. Flock makes idiomatic a novel and complicated semantics for automatically copying model elements from original to migrated model and a domain-specific language was preferred to repurposing an existing language to capture the semantics with a compact and tailored syntax. Section 5.4.1 discusses the principle tenets of Flock, which include user-defined migration rules and a novel algorithm for relating source and target model elements. In Section 5.4.2, Flock is demonstrated via application to three examples of model migration. Finally, Section 5.4.3 provides patterns and guidelines for using the model migration language provided by Flock to specify migration strategies. The work described in this section has been published in [Rose *et al.* 2010f], except for Section 5.4.3 which provides new material.

### 5.4.1 Design and Implementation

Flock has been designed to be a rule-based transformation language that mixes declarative and imperative parts. Consequently, Flock should be familiar to developers

---

[9] Forum discussion with Tom Morris, lead developer of the ArgoUML tool, `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required).

who have used hybrid-style M2M transformation languages, such as ATL and ETL [Kolovos *et al.* 2008b]. Flock is syntactically efficient, but semantically complex. In particular, the way in which Flock relates source to target elements is novel; it is neither a new- nor an existing-target relationship. Instead, elements are copied conservatively, as described below.

Like Epsilon HUTN (Section 5.2.4), Flock reuses parts of Epsilon (Section 2.3.2). In particular, Flock reuses EMC to provide interoperability with several modelling frameworks, and EOL for specifying the imperative part of user-defined migration rules.

### Abstract Syntax

As illustrated by Figure 5.22, Flock migration strategies are organised into modules (`FlockModule`). Flock modules inherit from EOL modules (`EolModule`) and hence provide language constructs for specifying user-defined operations and for re-using modules. Flock modules comprise any number of rules (`Rule`). Each rule has an original metamodel type (`originalType`) and can optionally specify a `guard`, which is either an EOL statement or a block of EOL statements. `MigrateRules` must specify an evolved metamodel type (`evolvedType`) and/or a `body` comprising a block of EOL statements.

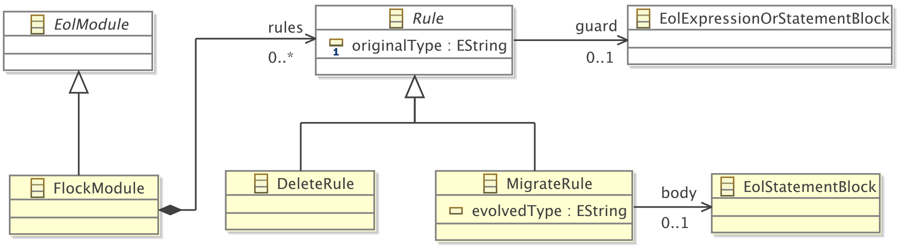


Figure 5.22: The abstract syntax of Flock.

### Concrete Syntax

Listing 5.21 shows the concrete syntax of migrate and delete rules. All rules begin with a keyword indicating their type (either `migrate` or `delete`), followed by the original metamodel type. Guards are specified using the `when` keywords. Migrate rules may also specify an evolved metamodel type using the `to` keyword and a `body` as a (possibly empty) sequence of EOL statements.

Note that Flock does not define a create rule. The creation of new model elements is instead encoded in the imperative part of a migrate rule specified on the containing type.

```
migrate <originalType> (to <evolvedType>)?
(when (:<eolExpression>)|({<eolStatement>+}))? {
  <eolStatement>*
}

delete <originalType>
(when (:<eolExpression>)|({<eolStatement>+}))?
```

Listing 5.21: Concrete syntax of migrate and delete rules.

### Execution Semantics

When executed, a Flock module consumes an original model, `O`, and constructs a migrated model, `M`. The transformation is performed in three phases: rule selection, equivalence establishment and rule execution. The behaviour of each phase is described below, and the first example in Section 5.4.2 demonstrates the way in which a Flock module is executed.

**Rule Selection** The rule selection phase determines an *applicable* rule for every model element, `e`, in `O`. As such, the result of the rule selection phase is a set of pairs of the form `<r,e>` where `r` is a migration rule.

A rule, `r`, is *applicable* for a model element, `e`, when the original type of `r` is the same type as (or is a supertype of) the type of `e`; and the guard part of `r` is satisfied by `e`.

The rule selection phase has the following behaviour:

- For each original model element, `e`, in `O`:
  - Identify for `e` the set of all applicable rules, `R`. Order `R` by the occurrence of rules in the Flock source file.
    - If `R` is empty, let `r` be a default rule, which has the type of `e` as both its original and evolved type, and an empty body.
    - Otherwise, let `r` be the first element of `R`.
  - Add the pair `<r,e>` to the set of selected rules.

Rules are ordered according to their position in the Flock source file. A rule that appears earlier (higher) in the source file has priority. Ordering rules according to generality is an alternative approach for distinguishing between applicable rules [Wallace 2005]. In model transformation, the generality of a rule might be assessed by considering its source type and guard. For Flock, the former approach was preferred, because it simplified implementation and is consistent with the way in which rules are selected in other languages of the Epsilon platform.

**Equivalence Establishment** The equivalence establishment phase creates an equivalent model element, `e'`, in M for every pair of rules and original model elements, `<r,e>`. The equivalence establishment phase produces a set of triples of the form `<r,e,e'>`, and has the following behaviour:

- For each pair `<r,e>` produced by the rule selection phase:
  - If `r` is a delete rule, do nothing.
  - If `r` is a migrate rule:
    - Create a model element, `e'`, in M. The type of `e'` is determined from the the `evolvedType` (or the `originalType` when no `evolvedType` has been specified) of `r`.
    - Copy the data contained in `e` to `e'` (using the *conservative copy* algorithm described in the sequel).
    - Add the triple `<r,e,e'>` to the set of equivalences.

**Rule Execution** The final phase executes the imperative part of the user-defined migration rules on the set of triples `<r,e,e'>`, and has the following behaviour:

- For each triple `<r,e,e'>` produced by the equivalence establishment phase:
  - Bind `e` and `e'` to new EOL variables named `original` and `migrated`, respectively.
  - Execute the body of `r` with EOL.

### Conservative Copy

Flock contributes a novel algorithm, termed *conservative copy*, that copies model elements from original to migrated model only when those model elements conform to the evolved metamodel. Conservative copy is a hybrid of the new- and existing-target source-target relationships that are commonly used in M2M transformation [Czarnecki & Helsen 2006], because some model elements are copied from source to target (as in an existing-target transformation), while some model elements must be copied explicitly (as in a new-target transformation).

Conservative copy operates on an original model element, `e`, and its equivalent model element in the migrated model, `e'`, and has the following behaviour:

- For each metafeature, `f` for which `e` has specified a value:
  - Find a metafeature, `f'`, of `e'` with the same name as `f`.
    - If no equivalent metafeature can be found, do nothing.
    - Otherwise, copy the original value (`e.f`) to produce a migrated value (`e'.f'`) if and only if the migrated value conforms to `f'`.

The definition of conformance varies over modelling frameworks. Typically, conformance between a value, `v`, and a feature, `f`, specifies at least the following constraints:

- The type of `v` must be the same as or a subtype of the type of `f`.
- The size of `v` must be greater than or equal to the lowerbound of `f`.
- The size of `v` must be less than or equal to the upperbound of `f`.

EMC provides drivers for several modelling frameworks, permitting management of models defined with EMF, the Metadata Repository (MDR), Z or XML. To support migration between metamodels defined in heterogeneous modelling frameworks, EMC has been extended to support conformance checking; each EMC driver provides conformance checking semantics specific to its modelling framework. Specifically, EMC define Java interfaces for specifying the way in which model values are written to a model, and an additional, conformance checking Java method has been added to the interface. When a Flock module is executed, conformance checking responsibilities are delegated to EMC drivers by calling the new method. The conformance checking support in EMC is applicable to other areas of the Epsilon platform and, in particular, could be used to add some static type checking to the Epsilon languages, which are currently dynamically typed.

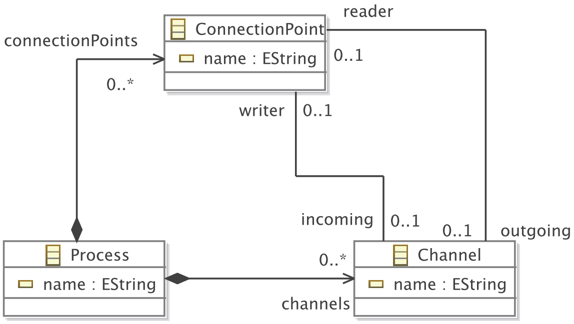
In response to some types of metamodel evolution, some categories of model value must be converted before being copied from the original to the migrated model. Again, the need for and semantics of this conversion varies over modelling frameworks. For example, reference values typically require conversion before copying because, once copied, they must refer to elements of the migrated rather than the original model. In this case, the set of equivalences (`<r,e,e'>`) can be used to perform the conversion. In other cases, the target modelling framework must be used to perform the conversion, such as when EMF enumeration literals are copied.

### 5.4.2 Examples of Flock Migration

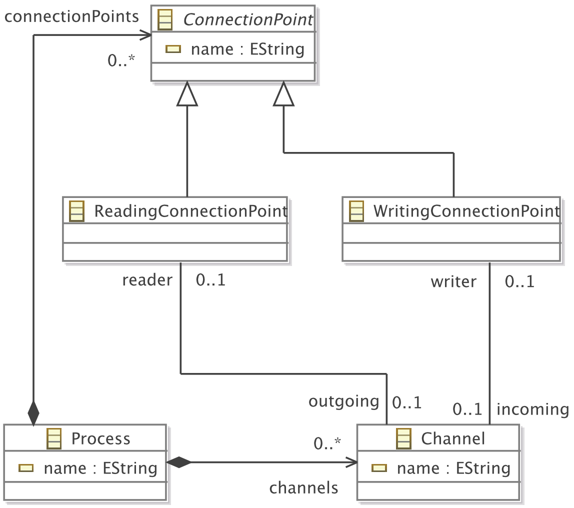
Flock is now demonstrated using three examples of model migration. The first example demonstrates the way in which a Flock module is executed and illustrates the semantics of conservative copy. The second describes the way in which the migration of the Petri net co-evolution example (Section 5.3.1) can be specified with Flock, and is included for direct comparison with the other languages discussed in Section 5.3. The final, larger example demonstrates all of the features of Flock, and is based on changes made to UML class diagrams between versions 1.5 and 2.0 of the UML specification.

#### Process-Oriented Migration in Flock

The first example considers the evolution of a process-oriented metamodel, introduced in Section 4.1.3 and described in Appendix B. The process-oriented metamodel was developed to explore the feasibility of a graphical model editor for representing programs written in process-oriented programming languages, such as occam-$\pi$ [Welch & Barnes 2005].



(a) Original metamodel.



(b) Evolved metamodel.

Figure 5.23: Evolution of the Process-Oriented metamodel (Appendix B)

The original metamodel, shown in Figure 5.23(a), has been evolved to distinguish between `ConnectionPoints` that are a reader for a `Channel` and `ConnectionPoints` that are a writer for a `Channel` by making `ConnectionPoint` abstract and introducing two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, as shown in Figure 5.23(b).

The model shown in Figure 5.24 conforms to the original metamodel in Figure 5.23(a) and is to be migrated. The model comprises three `Processes` named *delta*, *prefix* and

```
1  migrate ConnectionPoint to ReadingConnectionPoint when: original.outgoing.
       isDefined()
2  migrate ConnectionPoint to WritingConnectionPoint when: original.incoming.
       isDefined()
```

Listing 5.22: Redefining equivalences for the Component model migration.

*minus*; three `Channels` named *a*, *b* and *c*; and six `ConnectionPoints` named *a?*, *a!*, *b?*, *b!*, *c?* and *c!*.

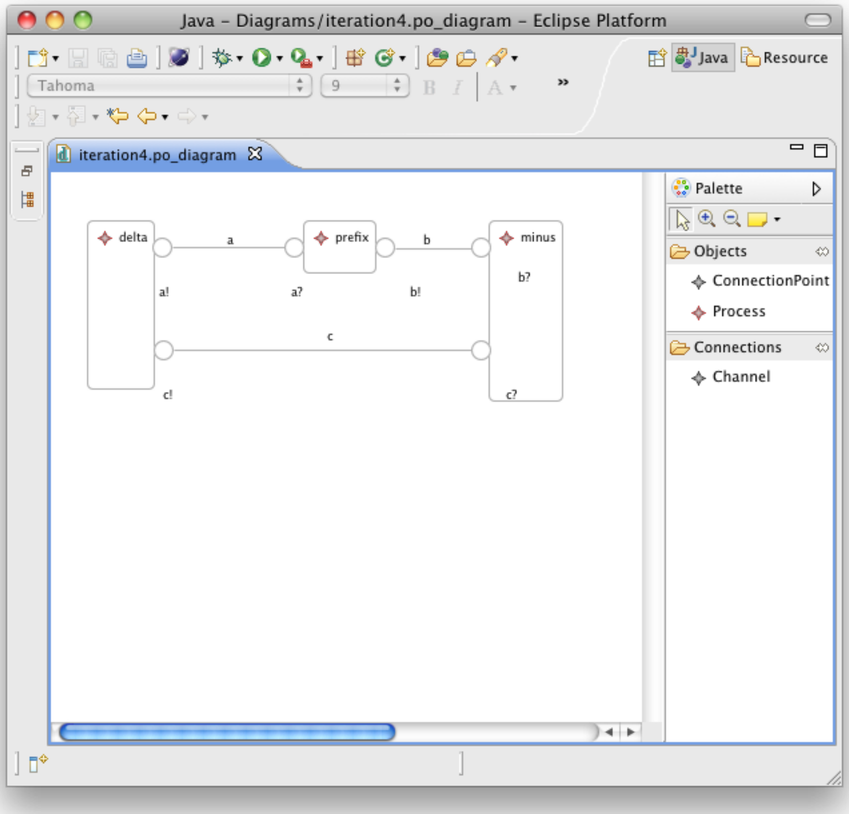


Figure 5.24: Process-Oriented model prior to migration

For the migration strategy shown in Listing 5.22, the Flock module will perform the following steps. Firstly, the rule selection phase produces a set of pairs `<r,e>`. For each `ConnectionPoint`, the guard part of the user-defined rules control which rule will be selected. `ConnectionPoints` `a!`, `b!` and `c!` have outgoing `Channels` (`a`, `b` and `c` respectively) and hence the migration rule on line 1 is selected. Similarly, the `ConnectionPoints` `a?`, `b?` and `c?` have incoming `Channels` (`a`, `b` and `c` respectively) and hence the migration rule on line 2 is selected. There is no `ConnectionPoint` with both an outgoing and an incoming `Channel`, but if there were, the first applicable rule (i.e. the rule on line 1) would be selected. For the other model elements (the `Processes` and `Channels`) no user-defined rules are applicable, and so default rules are used instead. A default rule has an empty body and identical original and evolved types. In other words, a default rule for the `Process` type is equivalent to the user-defined rule: `migrate Process to Process {}`

Secondly, the equivalence establishment phase creates an element, `e'`, in the migrated model for each pair `<r,e>`. For each `ConnectionPoint`, the evolved type of the selected rule (`r`) controls the type of `e'`. The rule on line 1 of Listing 5.22 was selected for the `ConnectionPoints` `a!`, `b!` and `c!` and hence an equivalent element of type `ReadingConnectionPoint` is created for `a!`, `b!` and `c!`. Similarly, an equivalent element of type `WritingConnectionPoint` is created for `a?`, `b?` and `c?`. For the other model elements (the `Processes` and `Channels`) a default rule was selected, and hence the equivalent model element has the same type as the original model element.

Finally, the rule execution phase performs a conservative copy for each original and equivalent model element in the set of triples `<r,e,e'>` produced by the equivalence establishment phase. The metamodel evolution shown in Figure 5.23 has not affected the `Process` type, and hence for each `Process` in the original model, conservative copy will create a `Process` in the migrated model and copy the values of all features. For each `Channel` in the original model, conservative copy will create an equivalent `Channel` in the migrated model and copy the value of the `name` feature from original to migrated model element. However, the values of the `reader` and `writer` features will not be copied by conservative copy because the type of these features has changed (from `ConnectionPoint` to `ReadingConnectionPoint` and `WritingConnectionPoint`, respectively). The values of the `reader` and `writer` features in the original model will not conform to the `reader` and `writer` features in the evolved metamodel. Finally, the values of the `name`, `incoming` and `outgoing` features of the `ConnectionPoint` class have not evolved, and hence are copied directly from original to equivalent model elements.

The rule execution phase also executes the body of each rule, `r`, for every triple in the set `<r,e,e'>`. The user-defined rules in Listing 5.22 have no body, and hence no further execution is performed in this case.

### Petri Nets Migration in Flock

The Petri net metamodel evolution demonstrates the core functionality of Flock. In Listing 5.23, `Nets` and `Places` are migrated automatically. Unlike the ATL migration strategy (Listing 5.18), no explicit copying rules are required. Compared to the COPE migration strategy (Listing 5.20), the Flock migration strategy does not need to unset the original `src` and `dst` features of `Transition`.

```
migrate Transition {
  for (source in original.src) {
```

```
  var arc := new Migrated!PTArc;
  arc.src := source.equivalent(); arc.dst := migrated;
  arc.net := original.net.equivalent();
 }

 for (destination in original.dst) {
  var arc := new Migrated!TPArc;
  arc.src := migrated; arc.dst := destination.equivalent();
  arc.net := original.net.equivalent();
 }
}
```

Listing 5.23: Petri nets model migration in Flock

### UML Class Diagram Migration in Flock

Figure 5.25 illustrates a subset of the changes made between UML 1.5 and UML 2.0. Only class diagrams are considered, and features that did not change are omitted. In Figure 5.25(a), association ends and attributes are specified separately. In Figure 5.25(b), the `Property` class is used instead. The Flock migration strategy (Listing 5.24) for Figure 5.25 is now discussed.

Firstly, `Attributes` and `AssociationEnds` are migrated to be `Properties` (lines 14 to 18, and 25 to 29). In particular, the `Association#navigableEnds` reference replaces the `AssociationEnd#isNavigable` attribute; following migration, each navigable `AssociationEnd` must be referenced via the `navigableEnds` feature of its `Association` (lines 26-28).

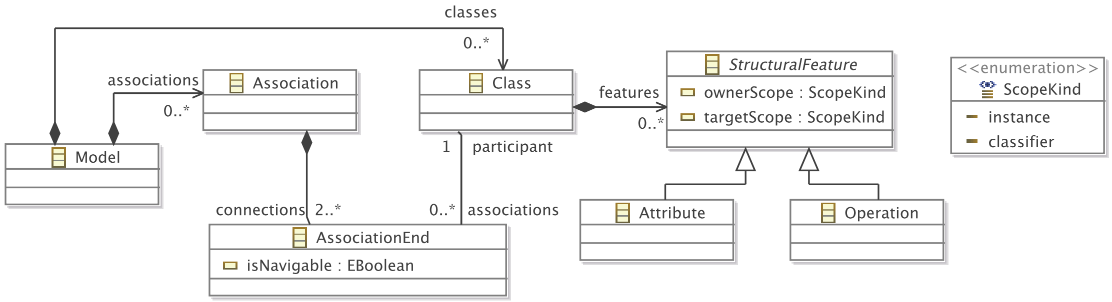
In UML 2.0, `StructuralFeature#ownerScope` has been replaced by `#isStatic` (lines 15-17 and 20-22). The UML 2.0 specification states that the UML 1.5 values `ScopeKind#classifier` and `#instance` should be migrated to true and to false, respectively.

The UML 1.5 `StructuralFeature#targetScope` feature is no longer supported in UML 2.0, and no migration path is provided. Consequently, line 14 deletes any model element whose `targetScope` is not the default value.

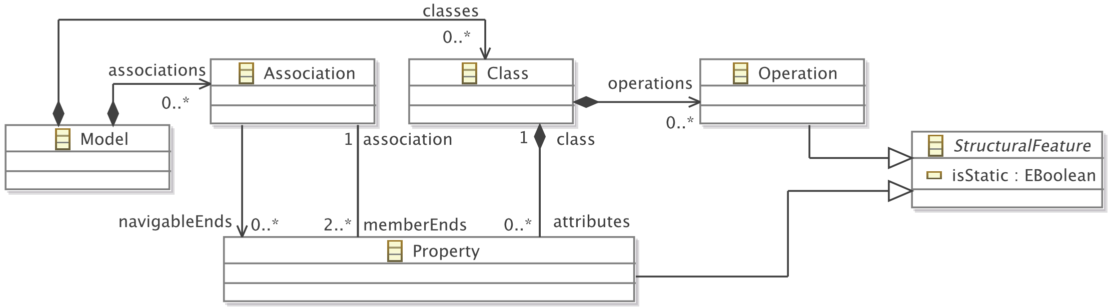
Finally, `Class#features` has been split to form `Class#operations` and `#attributes`. Lines 7 and 8 partition features on the original `Class` into `Operations` and `Propertys`. `Class#associations` has been removed in UML 2.0, and `AssociationEnds` are instead stored in `Class#attributes` (line 9).

```
migrate Association {
 migrated.memberEnds := original.connections.equivalent();
}

migrate Class {
 var fs := original.features.equivalent();
 migrated.operations := fs.select(f|f.isKindOf(Operation));
 migrated.attributes := fs.select(f|f.isKindOf(Property));
 migrated.attributes.addAll(original.associations.equivalent())
}

delete StructuralFeature when: original.targetScope <> #instance

migrate Attribute to Property {
 if (original.ownerScope = #classifier) {
  migrated.isStatic = true;
 }
}
migrate Operation {
 if (original.ownerScope = #classifier) {
  migrated.isStatic = true;
 }
}

migrate AssociationEnd to Property {
 if (original.isNavigable) {
  original.association.equivalent().navigableEnds.add(migrated)
 }
}
```

Listing 5.24: UML model migration in Flock



(a) Original, UML 1.5 metamodel.



(b) Evolved, UML 2.0 metamodel.

Figure 5.25: UML metamodel evolution

### 5.4.3 Developing Migration Strategies with Epsilon Flock

To demonstrate the way in which Flock can be used to specify migration strategies, this section provides a guide to the model migration language provided by Flock. In particular, the main features of EOL (the core language of Epsilon, which provides a foundation for Flock) are summarised, and then the way in which Flock can be used to specify model migration strategies in response to common types of metamodel evolution is discussed. Finally, guidelines are presented for specifying Flock migration strategies when the original and evolved metamodels use type inheritance, because the implementation of conservative copy provided by Flock affects the way in which such migration strategies are specified.

#### EOL for Migration Strategies

As discussed in Section 5.4.1, EOL is used to specify the guards and bodies of Flock rules. EOL is dynamically and strongly typed, and is a pure object-oriented language: primitives, collections and model elements are objects. The fundamental properties of EOL and its core functionality are now summarised.

**The `Any` type** The universal (top) type in EOL is called `Any`, and defines operations for performing null checks (`isDefined` and `isUndefined`), for type-checking (`isTypeOf` and `isKindOf`) and for printing to the standard output stream (`print` and `println`). In Flock, null and type checks are most often used in the guard of a migrate rule. For example, Listing 5.25 shows a fragment of a migration strategy that makes explicit that `LabourItems` are a special type of `JobItem` because they always reference an employee. Consequently, migration involves retyping only those `JobItems` that have a non-null value for the feature `employee` (i.e. `employee.isDefined()` evaluates to `true`).

**Built-in types** EOL provides four primitive types (`String`, `Integer`, `Real` and `Boolean`) and four collection types (`Bag`, `Sequence`, `Set` and `OrderedSet`). The built-in types provide many operations for manipulating primitive and collection values [Kolovos *et al.* 2006]. For example, Listing 5.26 demonstrates the way in which the `String` and `Sequence` types can be used to extract an author's name and email address from text of the form `"Joe Bloggs" <joe.bloggs@example.com>`. On line 2, the `String#split` operation is used to divide the original string into two parts (i.e. a `Sequence`). On lines 4 and 5 the `Collection#first` and `Collection#last` operations are used to access each part of the original string, and the `String#trim` operation is used to remove superfluous prefix and suffix characters from the substrings.

```
migrate JobItem to LabourItem when: original.employee.isDefined()
```

Listing 5.25: Using a non-null check to guard a migration rule

```
1  migrate Article {
2    var authorParts : Sequence := original.author.split(’<’);
3
4    migrated.authorName := authorParts.first.trim(’"’, ’"’);
5    migrated.authorEmail := authorParts.last.trim(’<’, ’>’);
6
7  }
```

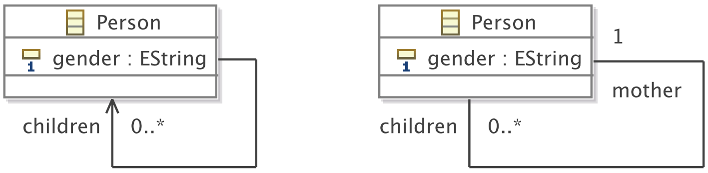
Listing 5.26: Using primitive and collection values

```
1  migrate Partition {
2    migrated.edges := original.contents.collect(e : Transition | e.equivalent());
3    migrated.nodes := original.contents.reject(ofs : ObjectFlowState | true).
         collect(n : StateVertex | n.equivalent());
4  }
```

Listing 5.27: Using higher-order operations on collections

Variable declarations are also demonstrated in Listing 5.26. Line 2 declares the `authorParts` variable, sets its type to `Sequence`, and sets its initial value. Variable declarations can omit the type (in which case `Any` is used) and the initialisation (in which case `null` is used). Variables are scoped to their enclosing context (denoted with brackets: {}). In Flock, a variable is local to either a migration rule or an operation, and the user cannot define global variables.

**Collection types** The collection types in EOL provide higher-order operations, such as `select` and `reject` (which filter a collection); `collect` (which applies a function to every element of a collection); and `exists` (which returns true iff at least one member of the collection satisfies the specified predicate). The higher-order operations have no side-effects: they do not modify the collection on which they are called. In Flock, the `select` and `collect` operators are most often used to partition values for migration in response to splitting a feature. For example, Listing 5.27 presents a fragment of a migration strategy for UML activity diagrams, which is discussed in full in Section 6.4. Briefly, the `Partition` metaclass has evolved such that the `contents` feature has been split into two features (`edges` and `nodes`). Migration involves dividing the elements of the `contents` into the new features. The `collect` operator is used to apply the built-in `equivalent()` operator to every `Transition` (line 2) and to every `StateVertex` (line 3) in a collection. Note that on line 3, the `reject` operator is used to filter out instances of `ObjectFlowState` from a collection. The higher-order operations are sometimes chained together as shown on line 3.



(a) Original metamodel. (b) Evolved metamodel.

Figure 5.26: Evolution of a unidirectional to a bidirectional reference.

```
migrate Person {
  migrated.mother := Original!Person.all
                         .selectOne(p|p.gender = "f" and
                                         p.children.includes(original))
                         .equivalent();
}
```

Listing 5.28: Using metamodel types

**Metamodel types** EOL programs can access metamodel types using the syntax `ModelName!MetamodelType`. Flock migration strategies are typically executed on two models (the original and migrated models), and conventionally the two models are called `Original` and `Migrated`. EOL provides metamodel type operations for accessing all instances of a type (`allInstances`) and for checking whether a type can be instantiated (`isInstatiable`). In Flock, metamodel types are sometimes used to navigate between model elements for which there exists no direct reference. For example, consider the metamodel evolution in Figure 5.26. The `children` reference has been evolved to become bidirectional. Migration involves setting the value of a `mother` reference for every `Person`. Searching the `children` reference for every `Person` in the model can be achieved using the `all` property (a syntactic shortcut for the `allInstances` operation) on the `Original!Person` metamodel type (Listing 5.28). (Note that `selectOne(x)` is a syntactic shortcut for `select(x).random()`).

**Instantiation with new** Like the primitive and collection types, metamodel types can be instantiated using the `new` keyword. Model elements can be deleted (completely removed from a model) using the `delete` keyword. Flock migration strategies conventionally have read-only access to the `Original` model and read-write access to `Migrated` model, and hence model elements are typically only created in – and deleted from – the `Migrated` model. Listing 5.29 demonstrates using the `new` keyword to create elements in the `Migrated` model. Specifically, one instance of `Connection` is created for every member of the `successor` reference for each `Block`.

```
migrate Block {
  for (successor in original.successors) {
    var connection := new Migrated!Connection;
    connection.source := migrated;
    connection.target := successor.equivalent();

    migrated.outgoing.add(connection);
  }
}
```

Listing 5.29: Creating new model elements

```
migrate Block {
  for (successor in original.successors) {
    var connection := connect(migrated, successor.equivalent());
    migrated.outgoing.add(connection);
  }
}

operation connect(source : Migrated!Block, target : Migrated!Block) : Connection {
  var connection := new Migrated!Connection;
  connection.source := source;
  connection.target := target;
  return connection;
}
```

Listing 5.30: Using a context-less custom operation

**Control flow** Listing 5.29 also demonstrates the use of the `for` construct for iterating over collection values. Additionally, EOL provides the `if` and `while` constructs for controlling the execution path of a program.

**Custom operations** Logic can be re-used and existing types enhanced via custom operations. For example, the migrate rule in Listing 5.29 can also be written using an operation to create instances of `Connection` (Listing 5.30). The `connect` operation can be re-used by other migrate rules.

The `connect` operation in Listing 5.30 is *context-less*: it is not called using dot notation. An existing type can be specified when defining an operation, and then the operation is invoked using dot notation. For example, the `connect` operation could be rewritten in the context of the `Migrated!Block` type (Listing 5.31). Compared to the context-less version, the custom operation is now invoked directly on the `migrated` object using dot notation. Note that the `self` built-in variable can be used to refer to the object on which the custom operation is invoked (line 10).

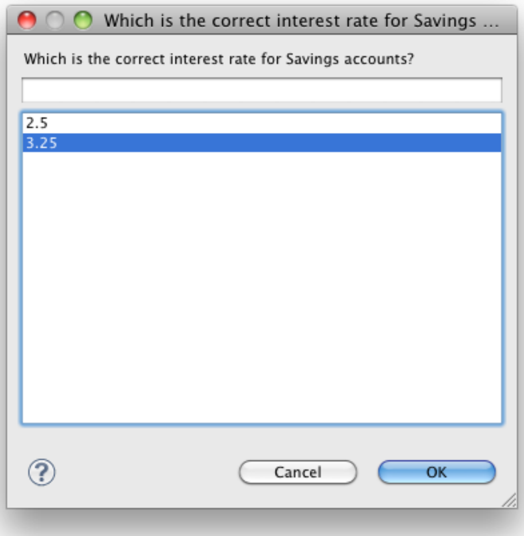
```
migrate Block {
  for (successor in original.successors) {
    var connection := migrated.connectTo(successor.equivalent());
    migrated.outgoing.add(connection);
  }
}

operation Migrated!Block connectTo(target : Migrated!Block) : Connection {
  var connection := new Migrated!Connection;
  connection.source := self;
  connection.target := target;
  return connection;
}
```

Listing 5.31: Using a custom operation in the context of a metamodel type

```
 1 migrate Account {
 2   if (migrated.type.interestRate <> original.interestRate) {
 3     var message := 'Which is the correct interest rate for ' + migrated.type.name
           + ' accounts?';
 4     var choices := Sequence{migrated.type.interestRate, original.interestRate};
 5     var defaultChoice := original.interestRate;
 6
 7     var chosen := System.getUserInput().choose(message, choices, defaultChoice);
 8
 9     // Do something with the interest rate selected by the user
10   }
11 }
```

Listing 5.32: Prompting for user input at runtime

**User input** Finally, EOL provides a mechanism for obtaining values from the user at runtime. The `System` built-in variable provides access to a user input library via its `getUser` operation. In Flock, user input can be used, for example, to resolve ambiguities during migration. For example, Listing 5.32 demonstrates the way in which the `choose` operation prompts the user to select between two possible interest rates for an account type (line 7). Epsilon provides several implementations of the user input interface for use in different contexts. For example, when running a migration strategy in Eclipse, a user is prompted with a graphical user interface (Figure 5.27). If a migration strategy is applied in a batch environment, the user might elect to specify input via Epsilon's command line interface for user input.



Which is the correct interest rate for Savings ...

Which is the correct interest rate for Savings accounts?

2.5
3.25

Figure 5.27: Epsilon graphical user interface for selecting between values at runtime.

### Specifying Migration Strategies with Epsilon Flock

Some common patterns for specifying migration strategies with Epsilon Flock are now presented. In particular, this section suggests migration strategy patterns that can used in response to some common types of metamodel evolution. The way in which rules are scheduled by the Flock engine and, in particular, the way in which elements are conservatively copied (Section 5.4.1) by the Flock engine influences the way in which migration is specified with Flock.

Recall that conservative copy automatically copies from the original to the migrated model only those elements that conform to the both the original and evolved metamodels. For example, model elements will not be automatically copied when their type does not exist in – or cannot be instantiated according to – the evolved metamodel. When a model element is conservatively copied from original to migrated model, its feature values are also conservatively copied. Like model elements, feature values that do not conform to the evolved metamodel are not automatically copied. The semantics of conservative copy is demonstrated with an example in Section 5.4.2.

**Changing feature values** The Flock engine executes migrate rules after conservative copy has been used to populate the migrated model. Consequently, migrate rules can be used to change – or even to delete – feature values that were set in the migrated model during conservative copying. Listing 5.33 demonstrates how to change a single-valued feature value (line 3), how to refer to the original value when changing a feature value (line 4) and how to add a value to a multi-valued feature value (line 5). Feature

```
 1 migrate Person {
 2   -- Changing model values
 3   migrated.name := 'Joe Bloggs';
 4   migrated.age := original.age + 1;
 5   migrated.luckyNumbers.add(42);
 6
 7   -- Unsetting model values
 8   migrated.address := null;
 9   migrated.lotteryNumbers.clear();
10 }
```

Listing 5.33: Changing and unsetting conservatively copied feature values

```
1 migrate Person {
2   if (original.id.isInteger()) {
3     migrated.id := original.id.asInteger();
4   } else {
5     ('Warning: no id has been set for ' + migrated.name).println();
6   }
7 }
```

Listing 5.34: Casting feature values

values are unset by either assigning null to the feature (single-valued features) as shown on line 8, or by calling the `Collection#clear` operation (multi-valued features) as shown on line 9.

**Casting feature values** When the type of a feature has been changed during metamodel evolution, it is sometimes necessary to explicitly convert a value from one type to another (such as from a `String` to an `Integer`). In EOL, primitive types can be cast using the built-in operations starting with `as` and `is`. Listing 5.34 demonstrates the use of the `isInteger` operation to check whether casting a `String` to an `Integer` will succeed (line 2), and the `asInteger` operation to perform the cast (line 3).

**Migrating reference values** Flock provides the `equivalent()` operation, which returns an element in the migrated model (or null) when invoked on an element of the original model, and can be used to migrate reference values. For example, invoking `equivalent()` on the built-in `original` variable will always return the value of the built-in `migrated` variable. Recall the migration rule in Listing 5.35, which was first introduced when describing the `new` keyword in EOL. The rule in Listing 5.35 iterates over a reference value in the original model, `original.successors` (line 2). The body of the loop has access to an element of the original model via the `successor` variable, and creates an instance of the `Connection` type in the migrated model.

```
1  migrate Block {
2    for (successor in original.successors) {
3      var connection := new Migrated!Connection;
4      connection.source := migrated;
5      connection.target := successor.equivalent();
6
7      migrated.outgoing.add(connection);
8    }
9  }
```

Listing 5.35: Using `equivalent()` to access migrated model elements

```
delete Block when: original.name = 'Foo';
```

Listing 5.36: Preventing the conservative copy of original model elements

Assigning the value of the `successor` variable (an element of the original model) to the `connection` (an element of the migrated model) will result in inter-model reference from the migrated model to the original model, which is typically incorrect in the context of model migration. Instead, the `equivalent()` operation is called on the `successor` to locate the element of the migrated model that corresponds to `successor`, which is an element of the original model (line 5).

**Creating model elements** When metamodel evolution has introduced new classes, migration often involves creating new model elements. With Flock, new model elements are created in the body of a migrate rule, because their values are either derived from data in the original model, or because they must be associated with another element of the migrated model. Listing 5.35 demonstrates the use of the `new` keyword to create an element in the migrated model (line 3). In this case, the newly created `Connection` is associated with the `Block` that is being migrated (line 4).

**Deleting model elements** Model migration sometimes involves deleting model elements, such as when metamodel evolution has restricted the set of conformant model elements. (For example, the UML 2.1.2 [OMG 2007a] disallows a type of `StructuralFeature` that was permitted in previous versions of the UML specification). In Flock, a `delete` rule can be used to prevent an element of the original model from being conservatively copied to the migrated model. Listing 5.36 demonstrates the use of a `delete` rule to ensure that instances of `Block` with the name `Foo` are not automatically copied from original to migrated model.

**Changing the type of model elements** Migration sometimes involves changing the type of existing model elements, such as when a metamodel class has been renamed or split. In Flock, a migrate rule optionally specifies a `to` part, which is used (during

```
1 migrate ConnectionPoint to ReadingConnectionPoint when: original.outgoing.isDefined()
2 migrate ConnectionPoint to WritingConnectionPoint when: original.incoming.isDefined()
```

Listing 5.37: Redefining equivalences for the Component model migration.

conservative copy) to determine the target type of an original model element. Listing 5.37 demonstrates[10] the use of two migrate rules to change the type of instances of `ConnectionPoint` to either `ReadingConnectionPoint` or `WritingConnectionPoint`. As discussed in Section 5.4.1, rules are prioritised from top to bottom in the Flock source file. Therefore, instances of `ConnectionPoint` that satisfy the guards of both rules in Listing 5.37 are retyped to `ReadingConnectionPoint` rather than `WritingConnectionPoint`, because the rule on line 1 takes precedence over the rule on line 2.

### Migration Strategies for Metamodels that use Inheritance

In addition to the Flock migration strategy patterns described above, users of Flock must consider one further facet of conservative copy that affect the way in which migration strategies with Flock should be specified. Specifically, conservative copy determines the target type of a model element by locating the first applicable migrate rule (from top to bottom in the source file). This has implications for the way in which migrate rules should be ordered, particularly when migrating models that conform to metamodel that use inheritance. The extent to which this implementation detail affects the efficacy of Flock is evaluated in Section 6.2.4. Here, the way in which migration strategies should be specified in response to the current implementation of conservative copy in Flock is discussed.

Consider the metamodel in Figure 5.28, which uses inheritance and is based on the GMF metamodel evolution described in Section C.3.1. Suppose that, during metamodel evolution, the type of the `figure` and `accessor` attributes is changed from string to integer, and migration involves deriving the migrated value for these features from the length of the original strings. Listing 5.38 demonstrates one way to specify this migration strategy with Flock. Note that, for the reasons given below, there is no rule specified on the `DiagramElement` type, and that the rule for the `DiagramLabel` type is placed above the rule for the supertype of `DiagramLabel`, `Node`.

Recall that the first applicable migrate rule is used to determine the target type. This has two implications for specifying the migration strategy with Epsilon Flock. Firstly, a migrate rule must not be specified for abstract types (such as `DiagramElement`) unless a concrete target type is specified via the to part of the migrate rule.

---

[10] The migrate rules in Listing 5.37 were first presented in Section 5.4.2 and are taken from the Flock migration strategy for the process-oriented example described in Appendix B).

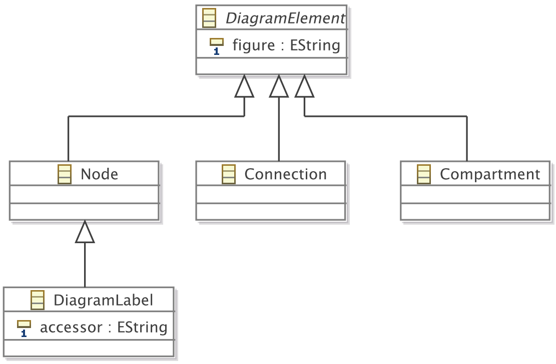


Figure 5.28: A metamodel that uses inheritance

```
migrate Compartment {
  migrated.figure := original.figure.length();
}

migrate Connection {
  migrated.figure := original.figure.length();
}

migrate DiagramLabel {
  migrated.figure := original.figure.length();
  migrated.accessor := original.accessor.length();
}

migrate Node {
  migrated.figure := original.figure.length();
}
```

Listing 5.38: A migration strategy for the metamodel evolution described in Figure 5.28

Hence, a separate migration rule must be specified for each subtype of `DiagramElement`.

Secondly, a migrate rule is applicable for every model element that instantiates its original type or, crucially, a subtype of its original type. Hence, a migrate rule specified for the `Node` type will also be applicable for instances of `DiagramLabel`. Consequently, the migrate rule for the `DiagramLabel` type must be placed above the the migrate rule for the `Node` type in the Flock source file. Otherwise, the `Node` rule will be applied to all instances of `Node` and `DiagramLabel`, and the migrate rule specific to `DiagramLabel` will never be applied.

The Flock migration strategy in Listing 5.38 exhibits some duplication due to the way in which migrate rules must be ordered for the current implementation of con-

```
migrate Compartment {
  migrated.migrateFigureFrom(original);
}

migrate Connection {
  migrated.migrateFigureFrom(original);
}

migrate DiagramLabel {
  migrated.migrateFigureFrom(original);
  migrated.accessor := original.accessor.length();
}

migrate Node {
  migrated.migrateFigureFrom(original);
}

operation Migrated!DiagramElement migrateFigureFrom(o : Original!DiagramElement)
      {
  self.figure := o.figure.length();
}
```

Listing 5.39: Using a custom operation to reduce duplication in the migration strategy in Listing 5.38

servative copy in Flock. To reduce the duplication, a custom operation can be used to encapsulate the repeated migration logic, as demonstrated in Listing 5.39. Custom operations were described above, when discussing the core functionality of EOL.

In summary, there are three guidelines to consider when using Flock to specify migration strategies for metamodels that use inheritance. Firstly, do not specify migrate rules for abstract types (unless a concrete type is specified via the to part of the migrate rule). Secondly, place rules for more specific types above rules for less specific types in the Flock source file. Finally, use custom operations to reduce duplication between rules that migrate parts of the same type hierarchy.

Using the guidelines and patterns described above, Flock can be used to specify migration strategies in response to metamodel evolution. Further examples of Flock migration strategies that use the patterns and guidelines described in this section can be found in Sections 6.2 and 6.4, and in Appendices B and C.

### 5.4.4 Summary

Requirements for a language tailored to model migration were described in Section 5.3. This section has presented Epsilon Flock, a language that seeks to fulfil those requirements. The way in which Flock has been designed and implemented has been discussed,

| **Tool** | **Automatic Copy** | **Automatic Unset** | **Modelling technologies** |
|---|---|---|---|
| Ecore2Ecore | ✓ | ✗ | XMI |
| ATL | ✗ | ✓ | EMF, MDR, KM3, XML |
| COPE | ✓ | ✗ | EMF |
| Flock | ✓ | ✓ | EMF, MDR, XML, Z |

Table 5.3: Properties of model migration approaches

with a particular focus on the novel source-target relationship, conservative copy (Section 5.4.1). Several examples of migration strategies constructed in Flock have been presented (Section 5.4.2). Further examples are given in Appendix C. Finally, the way in which metamodel developers can use the model migration language provided by Flock to specify migration strategies was discussed (Section 5.4.3).

Table 5.3 illustrates several characterising differences between Flock and the preexisting languages presented in Section 5.3. Due to its conservative copying algorithm, Flock is the only language to provide both automatic copying and unsetting. The evaluation presented in Section 6.2 explores the extent to which automatic copying and unsetting affect the conciseness of migration strategies.

All of the approaches considered in Section 5.3 support EMF. Both Flock and ATL support further modelling technologies, such as MDR and XML. However, ATL does not automatically copy model elements that have not been affected by metamodel changes. Therefore, migration between models of different technologies with ATL requires extra statements in the migration strategy to ensure that the conformance constraints of the target technology are satisfied. Because it delegates conformance checking to an EMC driver, Flock requires no such checks.

A more thorough examination of the similarities and differences between Flock and other migration strategy languages is provided by the evaluation presented in Chapter 6.

## 5.5 Chapter Summary

Three structures for identifying and managing co-evolution have been designed and implemented to approach the thesis requirements outlined in Chapter 4. The way in which modelling frameworks implicitly enforce conformance makes managing nonconformant models challenging, and the proposed metamodel-independent syntax (Section 5.1) extends modelling frameworks to facilitate the management of non-conformant models. The proposed textual modelling notation, Epsilon HUTN (Section 5.2), provides a human-usable notation as an alternative to XMI for performing user-driven co-evolution. Finally, Epsilon Flock (Section 5.4) contributes a domain-specific language for describing model migration.

The metamodel-independent syntax is a modelling framework extension that makes explicit the conformance relationship between models and metamodels. By binding models not to their metamodel but to a generic metamodel, the metamodel-independent syntax allows non-conformant models to be managed with modelling tools and model management operations. Furthermore, conformance checking is provided as a service, which can be scheduled at any time, and not just when models are loaded. The metamodel-independent syntax has been integrated with Concordance [Rose *et al.* 2010c] to provide a metamodel installation process that automatically reports conformance problems, and underpins the implementation of the second structure described in this chapter, a textual modelling notation.

For performing user-driven co-evolution, the textual modelling notation described in Section 5.2 provides an alternative to XMI. Unlike XMI, the notation introduced in this chapter implements the OMG standard for Human-Usable Textual Notation (HUTN) [OMG 2004] and is optimised for human usability. Epsilon HUTN, introduced here, is presently the sole reference implementation of HUTN. Constructing Epsilon HUTN by reusing the metamodel-independent syntax allows Epsilon HUTN to provide incremental and background conformance checking, and an XMI-to-HUTN transformation for loading non-conformant models. Section 6.1 explores the benefits and drawbacks of using the metamodel-independent syntax and Epsilon HUTN together to perform user-driven co-evolution.

The domain-specific language described in Section 5.4, Epsilon Flock, combines several concepts from existing model-to-model transformation languages to form a language tailored to model migration. In particular, Flock contributes a novel mechanism for relating source and target model elements termed conservative copy, which is a hybrid of new- and existing-target styles of model-to-model transformation. Flock extends and reuses Epsilon and hence interoperates transparently with several modelling technologies via EMC, the Epsilon Model Connectivity layer.

The metamodel-independent syntax, Epsilon HUTN, Epsilon Flock and Concordance have been released as part of Epsilon in the Eclipse GMT[11] project, which is the research incubator of arguably the most widely used MDE modelling framework, EMF. By re-using parts of Epsilon, the structures were implemented more rapidly than would have been possible when developing the structures independently. In particular, re-using EMC facilitated interoperability of Flock with several MDE modelling frameworks, which was exploited to manage a practical case of model migration in Section 6.4.

---

[11] `http://www.eclipse.org/gmt`

# Chapter 6

# Evaluation

Chapters 1 and 2 discussed the way in which contemporary MDE development environments provide structures and processes for creating and managing modelling artefacts. For MDE to be applicable in the large and to complex systems, however, MDE development environments must approach concerns relating to scalability, usability and managing evolutionary change. This chapter evaluates the structures and processes developed and described in this thesis with respect to the research hypothesis outlined in Section 1.4. In particular, the evaluation explores whether the structures and processes are effective for managing evolutionary change by assessing the extent to which they affect developer productivity in co-evolution.

As discussed in Chapter 1, the developed structures and processes are prototypes, and unlikely to be completely fit for industrial use in their current state. As such, the evaluation also aims to determine areas in which the proposed structures and processes might be usefully improved by identifying factors that affect their efficacy. The conclusions drawn in this chapter identify benefits and drawbacks of using the proposed structures and processes for managing co-evolution in contemporary MDE development environments and for industrial software-engineering projects.

Section 4.2.2 identified co-evolution management process, *user-driven co-evolution*, that had been used in real-world MDE projects and that had not been recognised in the literature. Chapter 5 described the implementation of two structures tailored for user-driven co-evolution, a *metamodel-independent syntax* and a *textual modelling notation*. Using a real-world example of user-driven co-evolution, Section 6.1 assesses the extent to which the dedicated structures proposed in Chapter 5 affect the productivity of user-driven co-evolution.

The remainder of the chapter evaluates *developer-driven co-evolution* (described in Section 4.2.3 and in which a migration strategy is specified in an executable format) and focuses on *Epsilon Flock* (Section 5.4), a transformation language tailored for model migration. Section 6.2 evaluates the novel source-target relationship strategy implemented in Flock, *conservative copy*, by comparison with two existing source-target relationship strategies using co-evolution examples from real-world projects. Sections 6.3 and 6.4 evaluate Flock as a whole, using an expert evaluation and a transformation contest, respectively. The transformation contest was judged by members of the MDE community, providing an opportunity for Flock and other transformation tools to be assessed in a peer review.

The evaluation described in Sections 6.3 and 6.4 was performed collaboratively, and the contributions of others are highlighted in those sections. The work presented in this chapter has been published in [Rose *et al.* 2010b, Rose *et al.* 2010d, Rose *et al.* 2010e].

## 6.1 Evaluating User-Driven Co-Evolution

Several real-world MDE projects in which user-driven co-evolution has been observed were reported in Chapter 4, and Chapters 3 and 4 highlighted that no tool support for user-driven co-evolution has yet been reported in the literature. To address this, Chapter 5 proposed two structures to support user-driven co-evolution, a metamodel-independent syntax (Section 5.1) and a textual modelling notation (Section 5.2). This section explores the extent to which the two structures increase the productivity of user-driven co-evolution, supporting the research hypothesis which stated that *integrating dedicated structures and processes with contemporary MDE environments is beneficial in terms of increased productivity.*

To explore the hypothesis, several approaches to evaluation could be used. The metamodel-independent syntax and textual modelling notation are freely available as part of Epsilon, a component of the Eclipse Modeling Project, so the productivity benefits of the structures could have been explored by gathering and analysing the opinion of users. However, this approach was discounted because drawing meaningful conclusions would have needed understanding of each user's domain, context and background. Evaluation could have been performed with a comprehensive user study that measured the time taken for developers to perform model migration with and without the dedicated structures for user-driven co-evolution. However, locating developers and co-evolution examples was not possible in the available time. Instead, evaluation was conducted by comparing two approaches to user-driven co-evolution using an example from a real-world MDE project. The first approach uses only the tools available in the Eclipse Modeling Framework (EMF); while the second approach uses EMF together with the metamodel-independent syntax and textual modelling notation introduced in Chapter 5.

Section 6.1.1 summarises Section 4.2.2, which described the challenges to productivity faced by developers while performing user-driven co-evolution with EMF. Section 6.1.2 introduces the example of user-driven co-evolution used to perform the evaluation. In Sections 6.1.3 and 6.1.4, the two approaches to user-driven co-evolution are demonstrated. The section concludes by comparing the two approaches and highlighting ways in which the metamodel-independent syntax and textual modelling notation increase developer productivity in the context of user-driven co-evolution.

### 6.1.1 Challenges for Performing User-Driven Co-Evolution

Two productivity challenges for performing user-driven co-evolution in contemporary MDE environments were identified in Section 4.2.2. Firstly, model storage representations are not optimised for use by humans, and so user-driven co-evolution – which typically involves changing models by hand – is made error-prone and time consuming. Secondly, the multi-pass parsers used to load models in contemporary MDE environments make user-driven co-evolution an iterative process, because not all conformance errors are reported in the first pass. The identification of these productivity challenges led to the derivation of the following research requirement in Section 4.3: *This thesis must demonstrate a user-driven co-evolution process that enables the editing of non-conformant models without directly manipulating the underlying storage representation and provides a conformance report for the original model and evolved metamodel.*

Two of the structures presented in Chapter 5 provide the foundation for fulfilling the above research requirement. The first, a metamodel-independent syntax, facilitates the conformance checking of a model against any metamodel. The second structure, the textual modelling notation *Epsilon HUTN*, allows models to be managed in a format that is reputedly easier for humans to use than XMI, the canonical model storage format [OMG 2004].

To fulfil the above research requirement, this section applies the metamodel-independent syntax and the textual modelling notation to demonstrate that user-driven co-evolution can be performed without encountering the challenges to productivity described above. To this end, an example of co-evolution is used to show the way in which user-driven co-evolution might be achieved with and without the metamodel-independent syntax and Epsilon HUTN.

### 6.1.2 Co-Evolution Example

The evaluation uses the co-evolution example taken from collaborative work with Adam Sampson, then a Research Associate at the University of Kent. The purpose of the collaboration was to build a prototypical editor for graphical models of programs written in process-oriented programming languages, such as occam-$\pi$ [Welch & Barnes 2005]. The graphical models would provide a standard notation for describing process-oriented programs. Part of the example was used to describe Epsilon Flock in 5.4.2.

The collaboration with Sampson was selected for the evaluation presented here for several reasons. Firstly, the work involved constructing a graphical model editor, a common MDE development activity [Amyot *et al.* 2006]. Secondly, the editor was developed in an incremental and iterative manner, and involved several different types of change to the metamodel, some of which affected conformance. Finally, a relatively small number of models were constructed during the collaboration, and hence a user-driven approach to managing co-evolution was more suitable than a developer-driven approach for this example.

The graphical model editor was developed using a MDE approach. A metamodel captures the abstract syntax of process-oriented programming languages, and code for a graphical model editor is automatically generated from the metamodel.

The final version of the graphical model editor is shown in Figure 6.1. The editor captures the three primary concepts used to specify process-oriented programs: processes, connection points and channels. Processes, represented as boxes in the graphical notation, are the fundamental building blocks of a process-oriented program. Channels, represented as lines in the graphical notation, are the mechanism by which processes communicate, and are unidirectional. Connection points, represented as circles in the graphical notation, define the channels on which a process can communicate. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel). Reading (writing) connection points are represented as white (black) circles in the graphical notation.

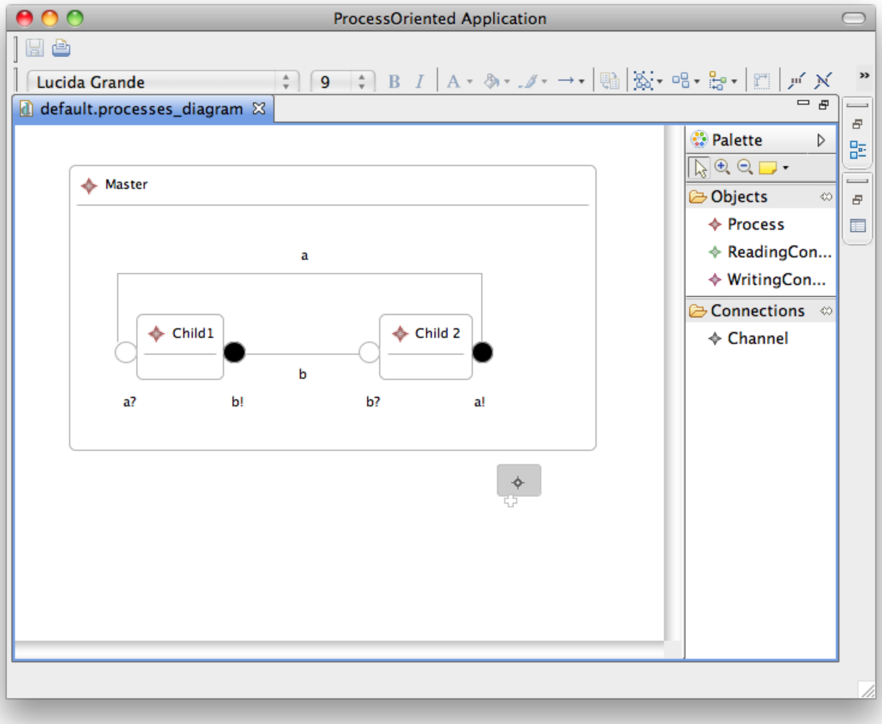


Figure 6.1: Final version of the prototypical graphical model editor.

The graphical model editor was implemented using EMF. The metamodel was specified in Ecore, the metamodelling language of EMF, and the graphical editor was generated from the metamodel using GMF. Section 2.3 describes in more detail the way in which EMF and GMF can be used to specify metamodels and to generate graphical model editors.

The process-oriented metamodel was developed iteratively, and the six iterations are described in Appendix B. During each iteration, the metamodel was changed. The evaluation described here uses an example of metamodel changes from the fifth iteration of the project. The way in which development proceeded during that iteration is described in Section B.5 and summarised below.

### Aim of Iteration 5

Iteration 5 of the process-oriented example was used to describe Epsilon Flock in 5.4.2. The purpose of the iteration was to refine the way in which connection points were represented. At the start of the iteration, the graphical model editor could be used to draw processes, channels and connection points. However, no distinction was made between reading and writing connection points.

Figure 6.2 shows a model represented in the graphical model editor before the iteration began. The model contains two processes (depicted as boxes), `P1` and `P2`, one channel (depicted as a line), `a`, and two connection points (depicted as circles), `a!` and `a?`.

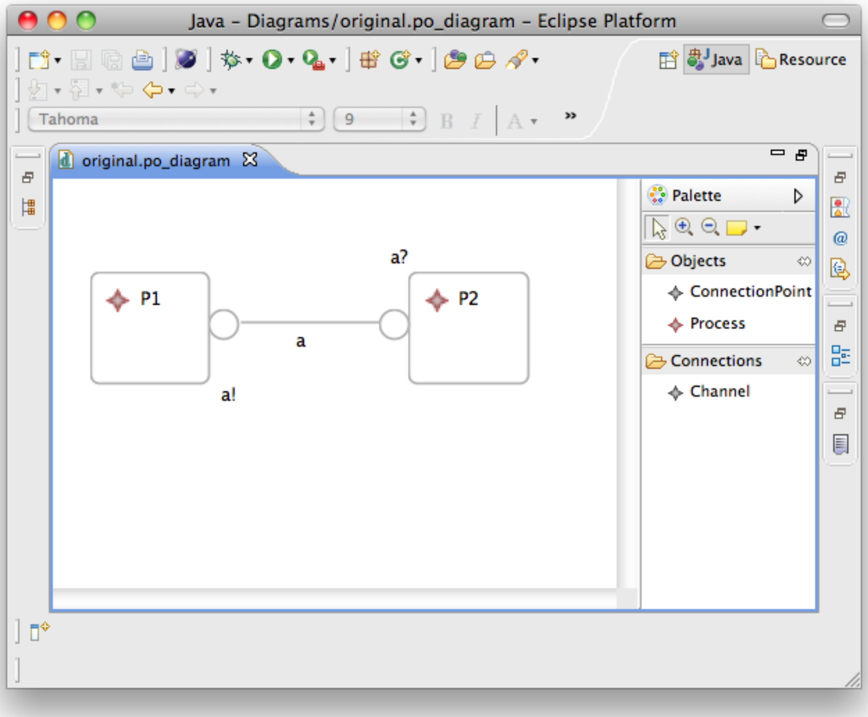


Figure 6.2: The graphical editor at the start of the iteration.

The aim of the iteration was to distinguish between reading and writing connection points in the graphical notation. The former are used to receive messages, and the latter to send messages. In Figure 6.2, `a?` is intended to represent a reading connection point, and `a!` a writing connection point. Sampson and I decided that the editor should be changed so that black circles would be used to represent writing connection points, and white circles to represent reading connection points. At the end of the iteration the model shown in Figure 6.2 would be represented as shown in Figure 6.3. Furthermore, the editor would ensure that `a?` was used only as the reader of a channel, and `a!` only as the writer of a channel. Before the iteration started, the editor did not enforce this constraint.

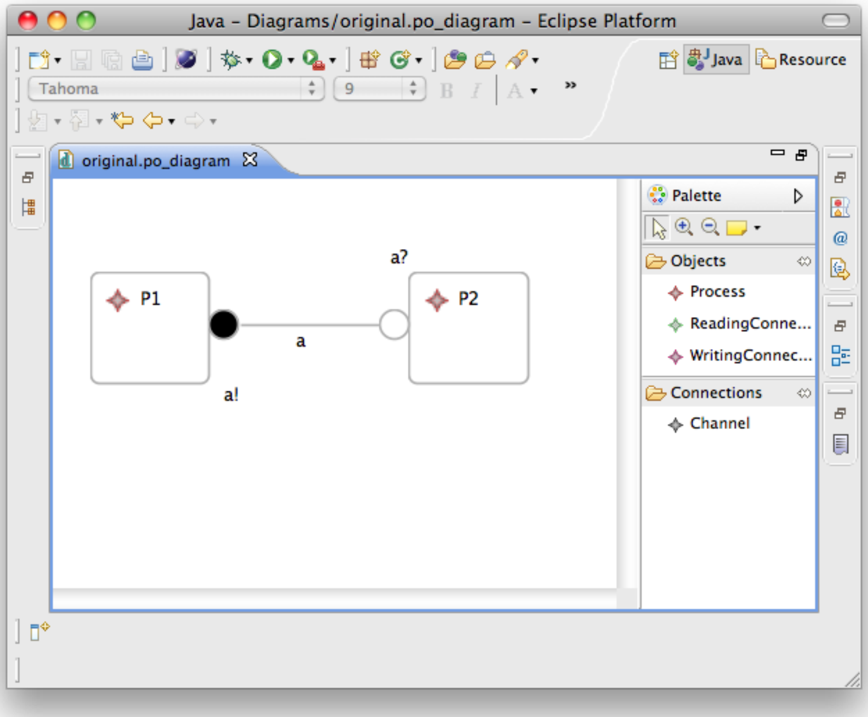
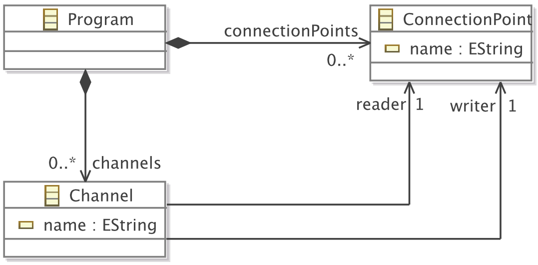


Figure 6.3: The graphical editor at the end of the iteration.

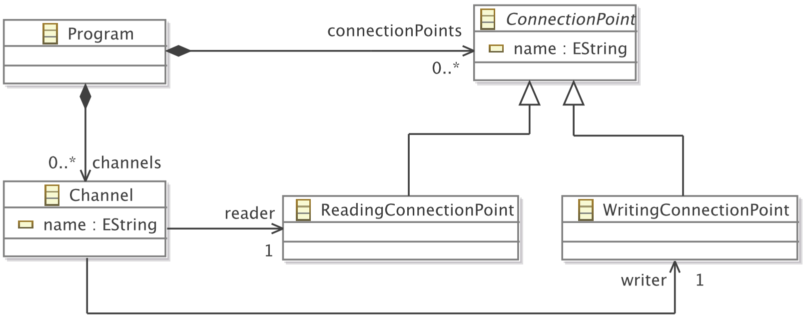
### Metamodel changes during Iteration 5

Before the iteration started, the metamodel, shown in Figure 6.4(a), did not distinguish between reading and writing `ConnectionPoint`s. A `ConnectionPoint` could be associated with a `Channel` via the `reader` or `writer` reference of `Channel`, but the type of a `ConnectionPoint` was not specified explicitly.

The way in which connection points were modelled was changed, resulting in the metaclasses shown in Figure 6.4(b). `ConnectionPoint` was made abstract, and two subtypes, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The `reader` and `writer` references of `Channel` were changed to refer to the new subtypes. The evolved metamodel correctly prevented the use of a `ConnectionPoint` as both a `reader` and a `writer`.



(a) Part of the original metamodel.



(b) Part of the evolved metamodel.

Figure 6.4: Process-oriented metamodel evolution.

Following the metamodel changes, a new version of the graphical editor was generated automatically from the metamodel using GMF. An annotation – not shown in Figure 6.4(b) – on the `WritingConnectionPoint` class was used to indicate to GMF that black circles were to be used to represent writing connection points in the graphical notation.

### Testing during Iteration 5

Testing the new version of the graphical editor highlighted the need for model migration. Attempting to load existing models, such as the one shown in Figure 6.2, caused an error because `ConnectionPoint` was now an abstract class. Any model specifying at least one connection point no longer conformed to the metamodel. Model migration was performed to re-establish conformance and to allow the models to be loaded.

Several models, presented in Appendix B, had been constructed when testing previous versions of the graphical editor. The models were used during each iteration to ensure that any changes had not introduced regressions. After the metamodel changes described above, the test models could no longer be loaded and required migration. A developer-driven co-evolution approach was not suitable for the development of process-oriented editor because only a few small models required migration in each iteration. A user-driven co-evolution approach was used instead, but, as no structures dedicated to user-driven co-evolution were available, co-evolution was performed by manually editing the storage representation of models.

The sequel describes the way in which migration was performed during the development of the process-oriented metamodel, without dedicated structures for performing user-driven co-evolution. Section 6.1.4 describes the way in which migration could have been performed using two of the structures presented in Chapter 5. The section concludes by comparing the two approaches.

### 6.1.3 User-Driven Co-Evolution with EMF

During the development of the process-oriented metamodel, no dedicated structures for performing user-driven co-evolution were available. Instead, migration was performed using only those tools available in EMF, as described below.

Migration with EMF involved identifying and fixing conformance errors, using the workflow shown in Figure 6.5. The workflow was first discussed in Chapter 4. When the user attempts to load a model, EMF automatically checks the conformance of the model. If the model does not conform to its metamodel, loading fails. To re-establish conformance, the user must edit by hand the underlying storage representation of the model, XMI. Recall that co-evolution is an iterative process because EMF uses a multi-pass XMI parser and cannot report all categories of conformance problem in the first pass.

One of the test models, shown in Figure 6.2, is now used to illustrate the way in which user-driven co-evolution was performed using the workflow shown in Figure 6.5. For the test model shown in Figure 6.2, the conformance problems shown in the bottom pane (and by the error markers in the left-hand margin of the top pane) of Figure 6.6 were reported by EMF. For example, the first conformance problem reported is shown in the tooltip in Figure 6.6, and states that a `ClassNotFoundException` was encountered because the "Class 'ConnectionPoint' is not found or is abstract."

The conformance problems were fixed by editing the XMI shown in Figure 6.6, producing the XMI shown in Figure 6.7. The type of each connection point element was changed to either `ReadingConnectionPoint` or `WritingConnectionPoint`. The former was used when the connection point was referenced via the `reader` reference of `Channel`, and the latter otherwise. The reconciled XMI is shown in Figure 6.7. On lines 4 and 7, the connection point model elements have been changed to include `xsi:type` attributes, which specify whether the connection point should instantiate `ReadingConnectionPoint` or `WritingConnectionPoint`.

Reconciling the conformance problems by editing the XMI required considerable knowledge of the XMI specification. For example, the `xsi:type` attribute is used to specify the type of the connection point model elements. In fact, it must be included for those model elements. However, for the other model elements in Figure 6.7 the `xsi:type` attribute is not necessary, and is omitted. When and how to use

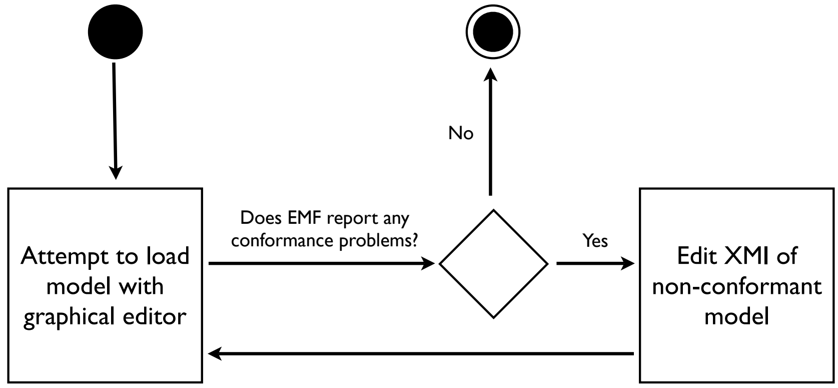
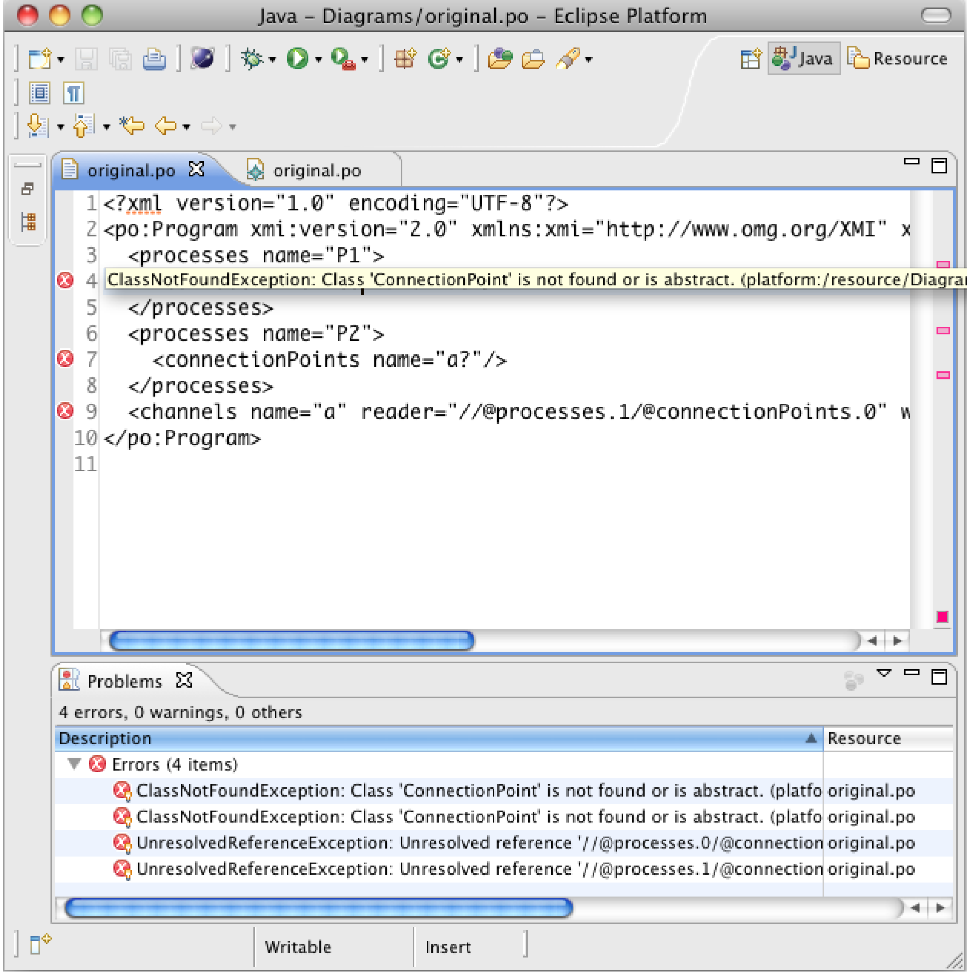


Figure 6.5: User-driven co-evolution with EMF

the `xsi:type` attribute is discussed further in the sidebar, in the XMI specification [OMG 2007c], and by the developers of EMF [Steinberg *et al.* 2008]. EMF abstracts away from XMI, and typically users do not interact directly with XMI. Therefore, it may be reasonable to assume that EMF users might not be familiar with XMI, and implementation details such as the `xsi:type` attribute.

During the development of the process-oriented editor, some mistakes were made when the XMI of the test models was edited by hand. For example, the wrong subtype of `ConnectionPoint` was used as the type of several connection point model elements. The mistake occurred because XMI identifies model elements using an offset from the root of the document. For example, consider the XMI shown in Figure 6.7. The channel on line 9 specifies the value "//@processes.1/@connectionPoints.0" for its `reader` attribute. The value is an XMI path referencing the first connection point ("@connectionPoints.0") contained in the second process ("@processes.1") of this document ("//"); in other words the connection point on line 7. One of Sampson's models contained many channels and connection points and incorrectly counting the connection points in the model led to several mistakes during the manual editing of the XMI. Each time a mistake was made when reconciling the XMI by hand, another loop around the workflow shown in Figure 6.5 was required.

As demonstrated above, migration using only the tools provided by EMF can be iterative and error-prone. The sequel demonstrates that, by using the dedicated structures described in Chapter 5, migration can be performed in one iteration, without requiring the developer to switch between conformance reporting and model migration tools. In addition, the sequel suggests how the mistake described above might be avoided by using Epsilon HUTN rather than XMI for manually migrating models.



```
<?xml version="1.0" encoding="UTF-8"?>
<po:Program xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" x
  <processes name="P1">
ClassNotFoundException: Class 'ConnectionPoint' is not found or is abstract. (platform:/resource/Diagra
  </processes>
  <processes name="P2">
    <connectionPoints name="a?"/>
  </processes>
  <channels name="a" reader="//@processes.1/@connectionPoints.0" w
</po:Program>
```

4 errors, 0 warnings, 0 others

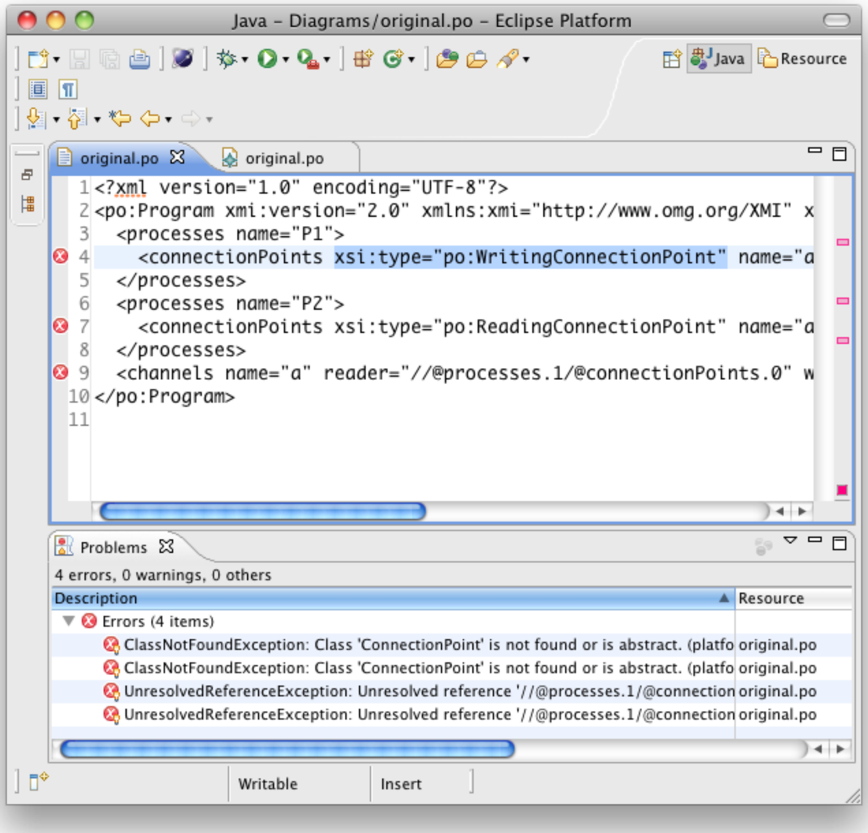
| Description | Resource |
|---|---|
| Errors (4 items) | |
| ClassNotFoundException: Class 'ConnectionPoint' is not found or is abstract. (platfo | original.po |
| ClassNotFoundException: Class 'ConnectionPoint' is not found or is abstract. (platfo | original.po |
| UnresolvedReferenceException: Unresolved reference '//@processes.0/@connection | original.po |
| UnresolvedReferenceException: Unresolved reference '//@processes.1/@connection | original.po |

Figure 6.6: XMI prior to migration

### 6.1.4 User-Driven Co-Evolution with Dedicated Structures

Chapter 5 describes two structures that can be used to perform user-driven co-evolution. Here, the functionality of the two structures, a metamodel-independent syntax and a textual modelling notation, is summarised. Subsequently, an approach that uses the metamodel-independent syntax and the textual modelling notation for migrating the model from the process-oriented example is presented. The model migration example presented in this section was performed retrospectively by the author after the process-oriented editor was completed, and demonstrates how migration might have been achieved with dedicated structures for user-driven co-evolution. The sequel compares the user-driven co-evolution approach presented in this section with the approach presented in Section 6.1.3.

The metamodel-independent syntax presented in Section 5.1 allows non-conformant



```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <po:Program xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" x
 3   <processes name="P1">
 4     <connectionPoints xsi:type="po:WritingConnectionPoint" name="a
 5   </processes>
 6   <processes name="P2">
 7     <connectionPoints xsi:type="po:ReadingConnectionPoint" name="a
 8   </processes>
 9   <channels name="a" reader="//@processes.1/@connectionPoints.0" w
10 </po:Program>
11
```

4 errors, 0 warnings, 0 others

| Description | Resource |
|---|---|
| Errors (4 items) | |
| ClassNotFoundException: Class 'ConnectionPoint' is not found or is abstract. (platfo | original.po |
| ClassNotFoundException: Class 'ConnectionPoint' is not found or is abstract. (platfo | original.po |
| UnresolvedReferenceException: Unresolved reference '//@processes.1/@connection | original.po |
| UnresolvedReferenceException: Unresolved reference '//@processes.1/@connection | original.po |

Figure 6.7: XMI after migration

**The `xsi:type` attribute**

In XMI, each model element must indicate the metaclass that it instantiates. Typically, the `xsi:type` attribute is used for this purpose. For example, the model element on line 4 of Figure 6.7 instantiates the metaclass named `WritingConnectionPoint`. To reduce the size of models on disk, the XMI specification allows type information to be omitted when it can be inferred. For example, line 9 of Figure 6.7 defines a model element that is contained in the `channels` reference of a `Process`. Because the `channels` reference can contain only one type of model element (`Channel`), the `xsi:type` attribute can be omitted, and the type information is inferred from the metamodel.

models to be loaded with EMF, and for the conformance of models to be checked against any metamodel. Epsilon HUTN, the textual modelling notation presented in Section 5.2 is underpinned by the metamodel-independent syntax and is an alternative to XMI for representing models in a textual format. Together, the two structures can be used for performing user-driven co-evolution using the workflow shown in Figure 6.8. The workflow was first discussed in Chapter 5. First, the user attempts to load a model in the graphical editor. If the model is non-conformant and cannot be loaded, the user clicks the "Generate HUTN" menu item, and the model is loaded with the metamodel-independent syntax and then a HUTN representation of the model is generated by Epsilon HUTN. The generated HUTN is presented in an editor that automatically reports conformance problems using the metamodel-independent syntax. The user edits the HUTN to reconcile conformance problems, and the conformance report is automatically updated as the user edits the model. When the conformance problems are fixed, XMI for the conformant model is automatically generated, and migration is complete. The model can then be loaded in the graphical editor.

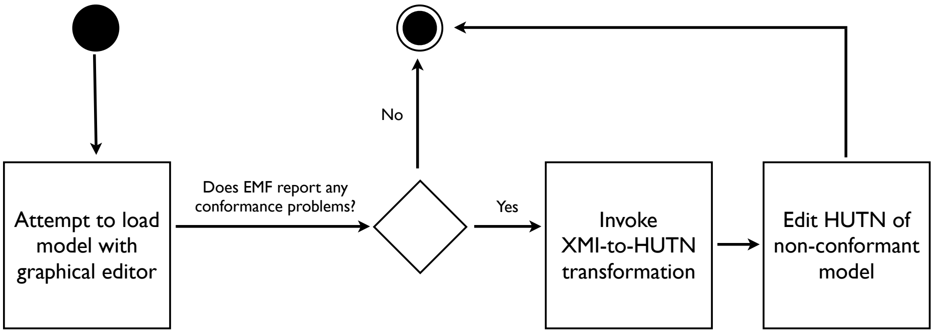
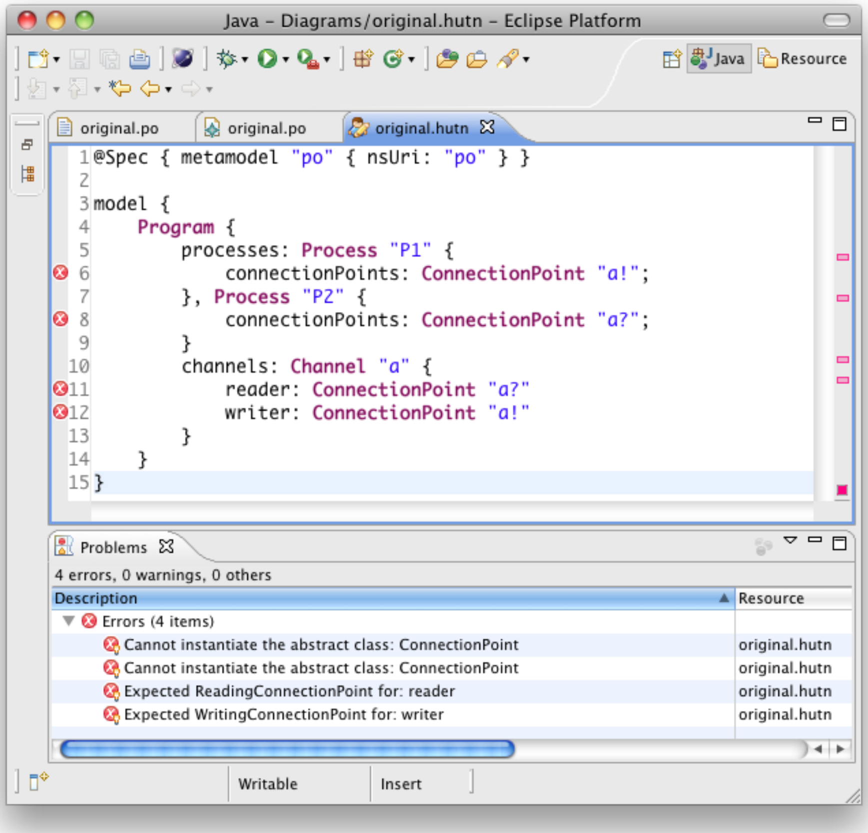


Figure 6.8: User-driven co-evolution with dedicated structures

The way in which the workflow shown in Figure 6.8 was used to perform user-driven co-evolution for the process-oriented metamodel is now demonstrated. For the model shown in Figure 6.2, the HUTN shown in Figure 6.9 was generated by invoking the automatic XMI-to-HUTN transformation. The HUTN development tools automatically present any conformance problems, as shown in the bottom pane (and the left-hand margin of the top pane) in Figure 6.9.

Conformance problems are reconciled manually by the user, who edits the HUTN source. Conformance is automatically checked whenever the HUTN is changed. For example, Figure 6.10 shows the HUTN editor when migration is partially complete. Some of the conformance problems have been reconciled, and the associated error-markers are no longer displayed in the left-hand margin.

When no conformance errors remain, Epsilon HUTN automatically generates XMI for reconciled model, and the user can now successfully load the migrated model with the graphical editor.



```
@Spec { metamodel "po" { nsUri: "po" } }

model {
    Program {
        processes: Process "P1" {
            connectionPoints: ConnectionPoint "a!";
        }, Process "P2" {
            connectionPoints: ConnectionPoint "a?";
        }
        channels: Channel "a" {
            reader: ConnectionPoint "a?"
            writer: ConnectionPoint "a!"
        }
    }
}
```

4 errors, 0 warnings, 0 others

| Description | Resource |
|---|---|
| Errors (4 items) | |
| Cannot instantiate the abstract class: ConnectionPoint | original.hutn |
| Cannot instantiate the abstract class: ConnectionPoint | original.hutn |
| Expected ReadingConnectionPoint for: reader | original.hutn |
| Expected WritingConnectionPoint for: writer | original.hutn |

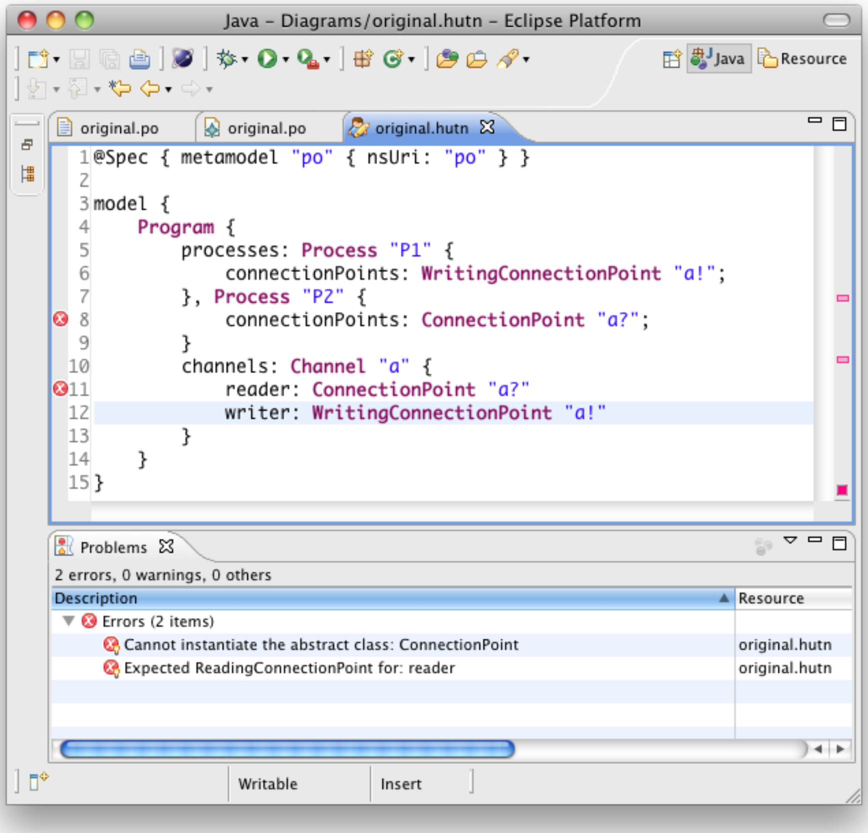
Figure 6.9: HUTN source prior to migration

### 6.1.5 Comparison

To suggest ways in which dedicated structures for user-driven co-evolution might increase developer productivity, the two user-driven co-evolution approaches demonstrated above are now compared. The first approach, described in Section 6.1.3, uses only those tools available in EMF for performing user-driven co-evolution, while the second approach, described in Section 6.1.4 uses two of the structures introduced in Chapter 5. Applying the approaches to the process-oriented example highlighted differences between the modelling notations used, and the way in which conformance problems were reported.

#### Differences in modelling notation

For reconciling conformance problems, the two approaches used different modelling notations, XMI and Epsilon HUTN. Differences in notation that might influence developer productivity during user-driven co-evolution are now discussed. However, further



```
@Spec { metamodel "po" { nsUri: "po" } }

model {
    Program {
        processes: Process "P1" {
            connectionPoints: WritingConnectionPoint "a!";
        }, Process "P2" {
            connectionPoints: ConnectionPoint "a?";
        }
        channels: Channel "a" {
            reader: ConnectionPoint "a?"
            writer: WritingConnectionPoint "a!"
        }
    }
}
```

2 errors, 0 warnings, 0 others

| Description | Resource |
|---|---|
| Errors (2 items) | |
| Cannot instantiate the abstract class: ConnectionPoint | original.hutn |
| Expected ReadingConnectionPoint for: reader | original.hutn |

Figure 6.10: HUTN source part way through migration

work is required to more rigorously explore the extent to which developer productivity is affected by the modelling notation, as discussed in Section 6.1.6.

The way in which the type of a model element is specified is different in XMI and HUTN. In XMI, type information can be omitted in some circumstances, but must be included in others. In HUTN, type information is mandatory for every model element. Consequently, every HUTN document contains examples of how type information should be specified, whereas XMI documents may not.

Reference values use paths in XMI (such as `//@processes.1/@connectionPoints.0`) and names (such as `a?`) in HUTN. XMI paths are constructed in terms of a document's structure and, as such, rely on implementation details. The name of a model element, on the other hand, is specified in the model, and does not rely on any implementation details. Consequently, it is conceivable that fewer mistakes will be made during user-driven co-evolution when reference values are specified by name rather than with the structural details of a model.

### Differences in conformance reports

The two approaches differ in the way in which conformance problems were reported, and, as a consequence, the first approach was iterative and the second was not. The differences might influence developer productivity during user-driven co-evolution. Again, further work is required to rigorously explore the extent to which developer productivity is affected by the differences in conformance reporting, as discussed in Section 6.1.6.

With EMF, user-driven co-evolution is an iterative process. Conformance errors are fixed by the user, who then reloads the reconciled model (with, for example, a graphical editor). Each time the model is loaded, further conformance problems might be reported when, for example, the user makes a mistake when reconciling the model. By contrast, the implementation of HUTN described in Section 5.2 uses a background compiler that checks conformance while the user edits the HUTN source. When the user makes a mistake reconciling the HUTN source, the error is reported immediately, and does not require the model to be loaded in the graphical editor.

Although not demonstrated here, user-driven co-evolution would, for some types of metamodel changes, remain an iterative process even if EMF performed conformance checking in the background. Because EMF uses a multi-pass parser, some types of conformance problem are reported before other types. For example, conformance problems relating to multiplicity constraints (e.g. a process does not specify a name, but name is a mandatory attribute) are reported after all other types of conformance problem. When several types of conformance problem have been affected by metamodel changes, user-driven co-evolution with EMF would remain an iterative process. Single-pass, background parsing is required to display all conformance problems while the user migrates a model.

## 6.1.6 Towards a more thorough comparison

Extensions to the evaluation of user-driven co-evolution are now discussed. In particular, alternative comparison methods might enable a more rigorous exploration of the ways in which dedicated structures for performing user-driven co-evolution influence developer productivity.

A comprehensive user study, involving hundreds of users, is one means for exploring the extent to which productivity varies when dedicated structures are used to perform user-driven co-evolution. Ideally, participants for the study would constitute a large and representative sample of the users of EMF. Productivity might be measured by the time taken to perform co-evolution. To remove a potential source of bias, several examples of co-evolution might be used.

Locating a reasonable number of participants and co-evolution examples for a comprehensive user study was not feasible in the context of this thesis. Nevertheless, the comparison presented in Section 6.1.5 suggests that productivity might be increased when using dedicated structures for user-driven co-evolution. By demonstrating an approach to user-driven co-evolution that uses dedicated structures, this thesis provides a foundation for further, more rigorous evaluation. For example, the HUTN specification

[OMG 2004] makes claims about the human-usability of the notation, but the usability of HUTN has not been studied or compared with other modelling notations. Epsilon HUTN (Section 5.2) is a reference implementation of HUTN and, as demonstrated by the evaluation presented here, facilitates the evaluation of HUTN and the comparison of HUTN to other modelling notations, such as XMI.

### 6.1.7 Summary

This section has demonstrated two approaches to user-driven co-evolution using a co-evolution example from a project in which a graphical model editor was created for process-oriented programs. The first approach used the structures available in EMF alone, while the second approach used two of the structures described in Chapter 5. Comparing the two approaches highlighted differences between the way in which conformance problems were reported and between the modelling notations used to reconcile conformance problems. The comparison described in Section 6.1.5 suggests that developer productivity might be increased by using the second approach, but, as discussed in Section 6.1.6, further work is required to more rigorously evaluate this claim.

## 6.2 Evaluating Conservative Copy

In contrast to the previous section, this section focuses on *developer-driven* co-evolution, in which migration is specified in a programming language. As discussed in Chapter 4, often a model-to-model (M2M) transformation language is used to specify migration. The M2M languages typically used to specify migration vary and, in particular, use different approaches to relating source and target model elements. This section evaluates the novel source-target relationship implemented in Flock (Section 5.4), *conservative copy*, by comparison to *new-target* and *existing-target* source-target relationships, which have been used for model migration in [Cicchetti *et al.* 2008, Garcés *et al.* 2009]) and in [Herrmannsdoerfer *et al.* 2009a, Hussey & Paternostro 2006]) respectively.

The evaluation performed in this section aims to demonstrate that migration strategies are more concise when written with a M2M language that uses conservative copy rather than when written with a M2M language that uses new- or existing-target. Arguably, more concise migration strategies lead to increased developer productivity in model migration. In manual specification approaches, migration strategies are written by hand and a concise language is likely to increase productivity because less code is written. In operator-based and inference approaches, migration strategies may be presented to the user to facilitate selection between feasible alternatives, and a concise language could increase productivity because less code must be read to comprehend a migration strategy. Other factors that might influence productivity, such as the comprehensibility of a migration strategy are not considered in this section, but in the sequel in which Epsilon Flock is evaluated by comparison with other co-evolution tools.

Conciseness might be measured in many ways. For instance, the number of lines of code have been used to argue that more concise software components indicate a high degree of inter-component re-use [Kolovos 2009]. In that context, the number of lines of code is an appropriate measure because the software components were written in a single programming language. The conciseness and understandability of programs can be approximated by determining the ratio of operators (language constructs) to operands (data) [Halstead 1977]. Halstead's Metrics are calculated from programming language constructs and, consequently, are affected by variations in programming languages. Here, counting lines of code and Halstead's Metrics are inappropriate because no single language implements the three styles of source-target relationship that are to be compared.

Instead, conciseness is measured by counting the frequency of *model operations*, atomic program statements that are used to manipulate the target (migrated) model. Model operations were specified in a language-independent manner and then mapped onto language-specific constructs to perform the counting. Therefore, the hypothesis for the comparison was: *specifying a migration strategy with conservative copy requires no more model operations than when new-target or when existing-target are used instead.* The results presented in Section 6.2.4 corroborate the hypothesis and highlight some limitations of the implementation of conservative copy in Flock.

The remainder of this section briefly recaps the theoretical differences between the three styles of source-target relationship (Section 6.2.1), describes the co-evolution examples and languages used in the comparison (Section 6.2.2), and details the comparison method (Section 6.2.3). Finally, the results of the comparison (Section 6.2.4) are used to support the claims made above, and to highlight limitations of the conservative copy implementation provided by Flock.

### 6.2.1 Styles of Source-Target Relationship

Two styles of source-target relationship, new-target and existing-target, are used in existing approaches to model migration, and a third is proposed in this thesis, conservative copy. The differences between the source-target relationships were discussed in Chapter 5 and are now summarised.

With a *new-target* source-target relationship, the migrated model is created afresh by the model migration strategy. The model migration language does not automatically copy any part of the original model to the migrated model. Consequently, any model elements that are not affected by metamodel evolution must be explicitly copied from original to migrated model.

With an *existing-target* source-target relationship, the migrated model is initialised as a copy of the original model. Prior to execution of the migration strategy, the migrated and original models are identical. Elements that no longer conform to the evolved metamodel might have been copied automatically from original to migrated model and, consequently, the migration strategy may need to delete model elements.

This thesis proposes a third style of source-target relationship termed *conservative copy*, which is a hybrid of new- and existing-target source-target relationships. Prior to the execution of the migration strategy, only those model elements that conform to the evolved metamodel are copied from original to migrated model

To compare the styles of source-target relationship, the evaluation uses a reference implementation of each style. The reference implementations are model-to-model transformation languages, which automatically copy model elements according to the style of source-target relationship. Flock, for example, provides a reference implementation of conservative copy.

### 6.2.2 Artefacts

To evaluate conservative copy, five examples of co-evolution taken from three projects, and three reference implementations of source-target relationships were used. The co-evolution examples and the selection process for the reference implementations are now discussed.

#### Co-evolution Examples

Conservative copy was designed in response to analysis of co-evolution examples, as discussed in Chapters 4 and 5. To reduce contamination of the comparison, the co-evolution examples used were distinct from those used to design conservative copy. The examples used for evaluating conservative copy are now summarised, and more details can be found in Appendix C.

Two examples were taken from the *Newsgroup* project, which performs statistical analysis of NNTP newsgroups, developed by Dimitris Kolovos, a lecturer in this department. One example was taken from changes made to *UML* (the Unified Modeling Language) between versions 1.4 [OMG 2001] and 2.2 [OMG 2007b] of the specification. Two examples were taken from *GMF* (Graphical Modeling Framework) [Gronback 2009], an Eclipse project for generating graphical model editors.

For the newsgroup and GMF projects, the co-evolution examples were identified from source code management systems. The revision history for each project was examined, and metamodel changes were located. The intended migration strategy was determined by speaking with the developer (for the Newsgroup project) and by examining examples and documentation (for GMF). The co-evolution example taken from UML was identified from the list of changes in the UML 2.2 specification [OMG 2007b], and by discussion with other UML users as described in Section 6.4.

For interoperability with the three reference implementations used in the comparison, the UML co-evolution was adapted. The original (UML 1.4 [OMG 2001]) metamodel is specified in XMI 1.2 [OMG 2007c], which is not supported by two of the reference implementations. The part of the UML 1.4 relating to activity graphs was reconstructed by the author in XMI 2.1 and used in place of the XMI 1.2 version. The reconstructed metamodel was checked by several UML users and was used in the expert evaluation described in Section 6.4, where the reconstructed metamodel is discussed further.

#### Reference Implementations Used in the Comparison

A formal semantics has not been specified for new-target, existing-target and conservative copy, and therefore the comparison reported in this section was performed using a reference implementation of each source-target relationship. Reference implementations for new- and existing-target were selected from the implementations used by existing approaches to model migration and compared to the implementation of conservative copy provided by Flock.

**New-target** As far as the author is aware, the Atlas Transformation Language (ATL) is the only new-target M2M transformation language that has been used for model migration. Existing new-target model migration approaches use higher-order transformations specified in ATL [Cicchetti *et al.* 2008, Garcés *et al.* 2009]. As such, ATL was used in the evaluation as the reference implementation of new-target.

**Existing-target** Two approaches to migration use existing-target transformations. In COPE [Herrmannsdoerfer *et al.* 2009a], migration strategies can be hand-written in Groovy when no co-evolutionary operator is applicable. COPE provides six Groovy functions for interacting with model elements, such as `set`, for changing the value of a feature, and `unset`, for removing all values from a feature. In this section, the term *Groovy-for-COPE* is used to refer to the combination of the Groovy programming language and the functions provided by COPE for use in hand-written migration strategies. In Ecore2Ecore [Hussey & Paternostro 2006], migration is performed when the original model is loaded, effectively an existing-target approach. For the comparison performed in this section, Groovy-for-COPE was preferred to Ecore2Ecore because the latter is not as expressive[1] and cannot be used for migration in the co-evolution examples considered here.

In summary, the comparison described in this section uses ATL for investigating new-target, Groovy-for-COPE for existing-target, and Flock for conservative copy.

### 6.2.3 Method

The comparison involved constructing migration strategies in each of the reference implementations (ATL, Groovy-for-COPE and Flock), identifying and counting model operations, and analysing the results. Following the selection of co-evolution examples and reference implementations, the author wrote a migration strategy for each co-evolution example in each of the reference implementations. The intended migration strategy was determined from models available in the source code management system of the co-evolution example (Newsgroup and GMF projects), or (for the UML example) by referring to the UML specification and discussing ambiguities with other UML users, as described in Section 6.4.

Next, a set of model operations were identified in a language independent manner and then mapped onto language constructs in ATL, Groovy-for-COPE and Flock. The counting of model operations was then automated by implementing a counting program,

---

[1] Communication with Ed Merks, Eclipse Modeling Project leader, 2009, available at `http://www.eclipse.org/forums/index.php?t=tree&goto=486690&S=b1fdb2853760c9ce6b6b48d3a01b9aac`

which was tested and used to further develop the comparison technique. Finally, the counting program was executed on the evaluation examples and the results investigated (Section 6.2.4).

Because the author is more familiar with Flock than with ATL and Groovy-for-COPE, the comparison method has an obvious drawback: the migration strategies written in the latter two languages might be more concise if they were written by the developers of ATL and Groovy-for-COPE. The evolutionary operators built into COPE provide many examples of migration strategy code written by the developer of COPE and, where possible, this code was re-used.

### Language-Independent Model Operations

The way in which model operations were identified and counted is now described. Four types of model operation were considered for inclusion in the evaluation: model element creation and deletion operators, and model value assignment and unassignment operators.

Creation and deletion operators are used to create or delete model elements in the migrated model. Assignment and unassignment operators are used to set or unset data values in the migrated model. Typically, assignment operators are used for copying values from the original to the migrated model.

Deletion and unassignment operators are not necessary when specifying model migration with new-target, because the migrated model is created afresh by the model migration strategy. Any deletion or unassignment would involve removing model elements or values created explicitly elsewhere in the migration strategy. By contrast, existing-target and conservative copy will automatically create model elements and assign model values prior to the execution of the model migration strategy and hence unassignment and deletion operators are required.

Creation operators were not included in the comparison because, unlike the other operators, they are difficult to specify with regular expressions (and hence automatically count). Moreover, in all of the co-evolution examples considered in the comparison, values are assigned to model elements after they are created. Consequently, at least one assignment operator is used whenever a creation operator would have been used.

### Model Operations in ATL, Groovy-for-COPE and Flock

The concrete syntax of the deletion, assignment and unassignment model operations in each language is now introduced. First however, it is important to note that the languages considered provide loop constructs and consequently a single model operation might be executed several times during the execution of a migration strategy. Here, a model operation is counted only once even if it is contained in a loop because the comparison is used to reason about the conciseness of migration strategies, and not about the way in which model operations are executed.

**New-target in ATL** For new-target in ATL, the following model operation was counted:

- **Assignment:**

  ```
  <feature> <- <value>
  ```

The assignment operator is used to copy values from the original to the migrated model. Typically, the `value` on the right-hand side is a literal, the value of a feature in the original model, or derived from a combination of the two. Listing 6.1 shows these typical uses of an assignment operator in ATL: line 4 assigns to a literal value, line 5 to the value of a feature in the original model, and line 6 to a value derived from two features in the original model that are separated with a literal value. In the listings in the remainder of this section, lines on which model operations appear are highlighted.

```
1  rule Person2Employee {
2   from o : Before!Person
3   to m : After!Employee (
4      role <- "Unknown",
5      id  <- o.id,
6      name <- o.forename + " " + o.surname
7    )
8  }
```

Listing 6.1: Assignment operators in ATL

As discussed above, deletion and unassignment operators are not used for new-target model migration.

**Existing-target in Groovy-for-COPE** For existing-target in Groovy-for-COPE, the following model operations were counted:

- **Assignment:**

  ```
  <element>.<feature> = <value>
  <element>.<feature>.add(<value>)
  <element>.<feature>.addAll(<collection_of_values>)
  <element>.set(<feature>, <value>)
  ```

- **Unassignment:**

  ```
  <element>.unset(<feature>)
  <element>.<feature>.remove(<value>)
  ```

- **Deletion:**

  ```
  delete <element>
  ```

Unlike ATL, Groovy-for-COPE provides distinct operators for assigning to single- and multi-valued features. The first assignment operator assigns to a single-valued feature, the second adds one value to a multi-valued feature, and the third adds multiple values to a multi-valued feature. The fourth form allows the feature name to be determined at runtime and, hence, facilitates reflective access to models.

COPE provides two forms of unassignment. The first can be used to unassign any feature. The second form is used to remove one value from a multi-valued feature.

**Conservative Copy in Epsilon Flock** Epsilon Flock, a transformation language tailored for model migration, was developed in this thesis and discussed in Chapter 5. The following model operations were counted:

- **Assignment:**

  ```
  <element>.<feature> := <value>
  <element>.<feature>.add(<value>)
  <element>.<feature>.addAll(<collection_of_values>)
  ```

- **Unassignment:**

  ```
  <element>.<feature> := null
  <element>.<feature>.remove(<value>)
  ```

- **Deleting:**

  ```
  delete <element>
  ```

The model operation listed above are the only mechanisms for changing the values or deleting model elements from the migrated model in a Flock migration strategy.

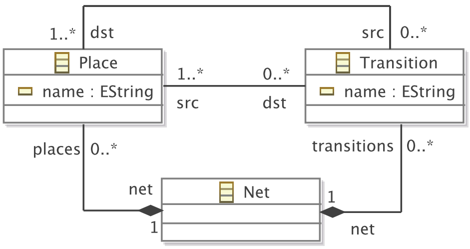
Like Groovy-for-COPE, Flock distinguishes between assignment to single-value and assignment to multi-valued features and, hence, provides three assignment operators. Unlike Groovy-for-COPE, Flock does not provide a form of assignment that allows the name of the assigned feature to be determined at runtime.

Flock does not provide a dedicated language construct for performing unassignment, which is instead achieved by assignment to `null`. One value can be removed from a multi-valued feature with the second form of unassignment.

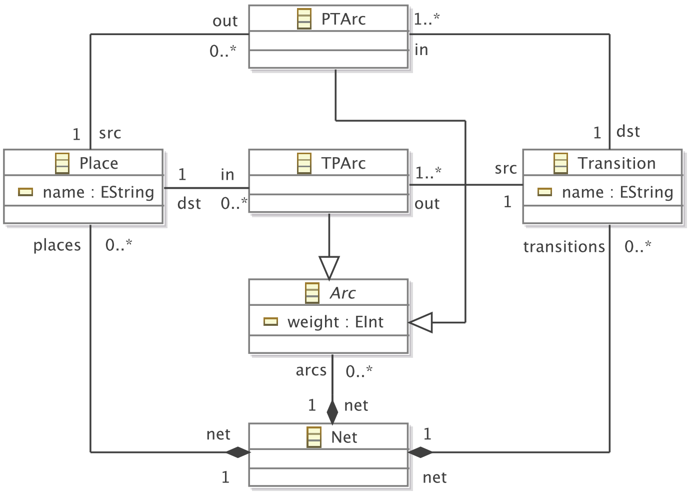
### Development and Testing of Method

The comparison method and a program for counting model operations were developed and tested by using the co-evolution examples described in Chapter 4, which were used to derive the thesis requirements. An example of model operation counting is given in the remainder of this section, along with the total number of model operations observed for each of the co-evolution examples described in Chapter 4.

Consider the example of metamodel-evolution shown in Figure 6.11. This is the Petri nets metamodel evolution described in Sections 5.3 and 5.4. The migration strategy replaces `Arcs` with `PTArcs` or `TPArcs`. In ATL, the migration strategy uses 12



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.11: Petri nets metamodel evolution. Taken from [Rose *et al.* 2010f].

model operations (Listing 6.2). In Groovy-for-COPE, the migration strategy uses 10 model operations (Listing 6.3) . In Flock, the migration strategy uses 6 model operations (Listing 6.4). These results are also shown in the *(Literature) PetriNets* row of Table 6.1.

Table 6.1 shows the total number of model operations needed to specify migration in ATL, Groovy-for-COPE and Flock for each of the co-evolution examples from Chapter 4. Because the examples used to produce the measurements shown in Table 6.1 were used to design Flock, they are not used to evaluate conservative copy. Instead, they are presented here to show the way in which the evaluation method was developed, and because one of the results (*Refactor: Change Ref to Cont*) highlighted a limitation of the existing-target and conservative copy implementations in COPE and Flock, which is discussed in Section 6.2.4.

```
rule Nets {
  from o : Before!Net
  to m : After!Net (
   places <- o.places,
   transitions <- o.transitions
  )
}

rule Places {
  from o : Before!Place
  to m : After!Place (
   name <- o.name
  )
}

rule Transitions {
  from o : Before!Transition
  to m : After!Transition (
    name <- o.name,
    "in" <- o.src->collect(p | thisModule.PTArcs(p,o)),
    out <- o.dst->collect(p | thisModule.TPArcs(o,p))
   )
}

lazy rule PTArcs {
  from place : Before!Place, destination : Before!Transition
  to ptarcs : After!PTArc (
    src <- place,
    dst <- destination,
    net <- destination.net
   )
}

lazy rule TPArcs {
  from transition : Before!Transition, destination : Before!Place
  to tparcs : After!TPArc (
    src <- transition,
    dst <- destination,
    net <- transition.net
   )
}
```

Listing 6.2: The Petri nets model migration in ATL

```
for (transition in Transition.allInstances) {
  for (source in transition.unset('src')) {
    def arc = petrinets.PTArc.newInstance()
    arc.src = source;
    arc.dst = transition;
    arc.net = transition.net
  }

  for (destination in transition.unset('dst')) {
    def arc = petrinets.TPArc.newInstance()
    arc.src = transition;
    arc.dst = destination;
    arc.net = transition.net
  }
}

for (place in Place.allInstances) {
  place.unset('src');
  place.unset('dst');
}
```

Listing 6.3: The Petri nets model migration in Groovy-for-COPE

```
migrate Transition {
  for (source in original.src) {
    var arc := new Migrated!PTArc;
    arc.src := source.equivalent();
    arc.dst := migrated;
    arc.net := original.net.equivalent();
  }

  for (destination in original.dst) {
    var arc := new Migrated!TPArc;
    arc.src := migrated;
    arc.dst := destination.equivalent();
    arc.net := original.net.equivalent();
  }
}
```

Listing 6.4: Petri nets model migration in Flock

|  | **Migration Language**<br>Source-Target Relationship |  |  |
|---|---|---|---|
| (Project) Example | **ATL**<br>New | **G-f-C**<br>Existing | **Flock**<br>Conservative |
| (FPTC) Connections | 6 | 6 | 3 |
| (FPTC) Fault Sets | 7 | 5 | 3 |
| (GADIN) Enum to Classes | 4 | 1 | 0 |
| (GADIN) Partition Cont | 5 | 3 | 2 |
| (Literature) PetriNets | 12 | 10 | 6 |
| (Process-Oriented) Split CP | 8 | 1 | 1 |
| (Refactor) Cont to Ref | 4 | 5 | 3 |
| (Refactor) Ref to Cont | 3 | 5 | 3 |
| (Refactor) Extract Class | 5 | 4 | 2 |
| (Refactor) Extract Subclass | 6 | 0 | 0 |
| (Refactor) Inline Class | 4 | 5 | 2 |
| (Refactor) Move Feature | 6 | 2 | 1 |
| (Refactor) Push Down Feature | 6 | 0 | 0 |

Table 6.1: Model operation frequency (analysis examples).

|  | **Migration Language**<br>Source-Target Relationship |  |  |
|---|---|---|---|
| (Project) Example | **ATL**<br>New | **G-f-C**<br>Existing | **Flock**<br>Conservative |
| (Newsgroup) Extract Person | 9 | 4 | 3 |
| (Newsgroup) Resolve Replies | 8 | 3 | 2 |
| (UML) Activity Diagrams | 15 | 15 | 8 |
| (GMF) Graph | 101 | 10 | 13 |
| (GMF) Gen2009 | 310 | 16 | 16 |

Table 6.2: Model operation frequency (evaluation examples).

### 6.2.4 Results

By counting the model operations in model migration strategies, the similarities and differences between the three styles of source-target relationship were investigated. The five co-evolution examples discussed in Section 6.2.2 were measured to obtain the results shown in Table 6.2.

The comparison hypothesis stated that *specifying a migration strategy with conservative copy requires no more model operations than when new-target or when existing-target are used instead.* For four of the five examples in Table 6.2, the results support the hypothesis, but the results for the GMF Graph example do not.

The comparison hypothesis did not consider differences between new-target and existing-target, but the results show that, for the most part, a migration strategy uses fewer model operations when using existing-target rather than new-target. For all of the examples in Table 6.2 and most of the examples in Table 6.1, no migration strategy specified with existing-target contained fewer model operations when specified with new-target. However, three of the Refactor examples in Table 6.1 required more model operations when specified with existing-target than when specified with new-target.

The results are now investigated, starting by discussing the way in which the results support the comparison hypothesis. Subsequently, results that contradict the hypothesis are investigated. Two limitations of the conservative copy implementation in Flock were discovered via the investigation of results.

### Investigation of results

As discussed in Section 6.2.1, new-target, existing-target and conservative copy initialise the migrated model in a different way. New-target initialises an empty model, while existing-target initialises a complete copy of the original model. Conservative copy initialises the migrated model by copying only those model elements from the original model that conform to the migrated metamodel.

For four of the co-evolution examples, the results in Table 6.2 support the comparison hypothesis, which stated that *specifying a migration strategy with conservative copy requires no more model operations than when new-target or when existing-target are used instead*. Additionally, the results in Table 6.2 indicate that a migration strategy can be specified with fewer model operations when using existing-target rather than new-target. In particular, for the GMF examples shown in Table 6.2, evolution affected only a small proportion of the metamodel, and the ATL (new-target) migration strategies use many more model operations than Groovy-for-COPE (existing-target) and Flock (conservative copy).

This result can be explained by considering how new-target differs from exiting-target and conservative copy when the source (original) and target (evolved) metamodels are very similar. New-target initialises an empty model and, hence, every element of the migrated model must be derived from the original model. For model elements that do not need to be changed in response to metamodel evolution, the migration strategy must copy those elements without change. For instance, the new-target version of the GMF Graph and Gen migration strategies contain many transformation rules such as the one shown in Listing 6.5, which exist only for copying model elements from the original to the migrated model. In Listing 6.5, 5 model operations are used (all assignments) to copy values from the original to the migrated model. The features shown in Listing 6.5 (`figures`, `nodes`, `connections`, `compartments` and `labels`) were not changed during metamodel evolution. Unlike new-target, existing-target and conservative copy do not require explicit copying of model elements from the original to migrated model due to the way in which they initialise the migrated model.

In the UML co-evolution example (Table 6.2) and the Refactor Inline Class (Table 6.1), a large proportion of metamodel features were renamed. For these examples, expressing migration with an existing-target transformation language requires more

```
rule Canvas2Canvas {
  from o : Before!Canvas
  to m : After!Canvas (
    figures <- o.figures,
    nodes <- o.nodes,
    connections <- o.connections,
    compartments <- o.compartments,
    labels <- o.labels
  )
}
```

Listing 6.5: An extract of the GMF Graph model migration in ATL

model operations than using a new-target transformation language. Existing-target requires two model operations be used when a feature is renamed, while new-target and conservative copy require only one model operation. For instance, the `transitions` feature of `ActivityGraph` was renamed to `edge` in the UML co-evolution example. The code used for migration in response to this change for new-target, existing-target and conservative copy is shown below.

**New-target:** `edge <- transitions`
**Existing-target:** `element.edge = element.unset(transitions)`
**Conservative copy:** `migrated.edge := original.transitions`

As shown above, migration in response to feature renaming typically requires one model operation when using new-target and conservative copy (an assignment). When using existing-target, the equivalent migration strategy requires an additional model operation (an unassignment) that removes the value from the old feature. Note that, in Groovy-for-COPE, the `unset` function unassigns a feature and returns the (unassigned) value.

The results in Table 6.2 support the comparison hypothesis for four of the five examples. When specified with conservative copy, the migration strategies did not contain explicit copying (which was required when using new-target for the GMF examples) and used one rather than two model operations for migration in response to feature renaming (which required two model operations when using existing-target). However, the GMF Graph co-evolution example does not support the hypothesis due to a limitation of the way in which conservative copy is implemented in Flock. This limitation is described in the sequel.

Two conclusions can be drawn from investigating the results of the comparison. Firstly, in general, fewer model operations are used when specifying a migration strategy with a conservative copy migration language than when specifying the same migration strategy with a new- or existing-target migration language. Secondly, in the examples studied here, there are often more features unaffected by metamodel evolution than affected. Consequently, specifying model migration with a new-target migration language requires more model operations than in an existing-target migration language for the examples shown in Tables 6.1 and 6.2. It has been suggested that metamodel evolution often involves changes to relatively few metamodel elements [Sprinkle 2003], and the results presented in this section support this hypothesis.

### Limitation 1: Duplication when migrating subtypes

For the GMF Graph example (Table 6.2), conservative copy requires more model operations than existing-target. Investigation of this result revealed a limitation of the conservative copy implementation provided by Flock, which is now described and illustrated using a simplified fragment of the GMF Graph co-evolution example.

Figure 6.12 shows[2] part of the GMF Graph metamodel prior to evolution, which has been simplified for illustrative purposes. In the real metamodel, the `figure` and `accessor` features are references to other metamodel classes, rather than attributes. When the metamodel evolved, the types of the `figure` and `accessor` features were changed. Here, let us assume that their types were changed from string to integer. The real metamodel changes are described in Section C.3.1.

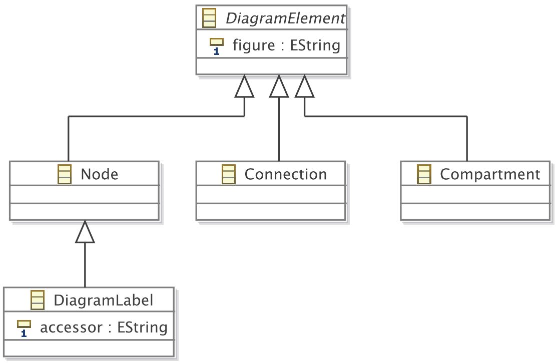


Figure 6.12: Simplified fragment of the GMF Graph metamodel.

In response to re-typing of the `figure` and `accessor` features, the migration strategy derived new values for the `figure` and `accessor` features. In the real example, a new model element was created and used to decorate [Gamma *et al.* 1995] each old value. In the simplified example presented here, the new integer value will be derived from the old string value by using its length. Section C.3.1 presents the strategies used to perform migration for the real metamodel changes.

As demonstrated below, ATL and Groovy-for-COPE provide mechanisms for reusing migration code between subtypes. Migration of the `figure` feature can be

---

[2] Note that this metamodel was also used to discuss specifying migration strategies with Flock in Section 5.4.3.

```
1  abstract rule DiagramElements {
2    from o : Before!DiagramElement
3    to  m : After!DiagramElement (
4      figure <- o.figure.length()
5    )
6  }
7
8  rule Nodes extends DiagramElements {
9    from o : Before!Node
10   to  m : After!Node
11 }
12
13 rule Connections extends DiagramElements {
14   from o : Before!Connection
15   to  m : After!Connection
16 }
17
18 rule Compartments extends DiagramElements {
19   from o : Before!Compartment
20   to  m : After!Compartment
21 }
22
23 rule DiagramLabels extends Nodes {
24   from o : Before!DiagramLabel
25   to  m : After!DiagramLabel (
26     accessor <- o.accessor.length()
27   )
28 }
```

Listing 6.6: Simplified GMF Graph model migration in ATL

specified once and used for migrating all subtypes of `DiagramElement`. Currently, Flock does not provide a mechanism for re-using migration code between subtypes.

In ATL (Listing 6.6), the GMF Graph migration strategy was expressed using two model operations: the two assignment operations on lines 4 and 26. For `Nodes`, `Connections` and `Compartments`, migration of the `figure` feature is achieved by extending the `DiagramElement` transformation rule. Note the use of the `extends` keyword on lines 8, 13 and 18 for inheriting the rule on lines 1-4. For `DiagramLabels`, the values of both the `accessor` and `figure` features must be migrated. On lines 23-28, the `DiagramLabels` rule extends the `Nodes` rule (and hence the `DiagramElements` rule) to inherit the body of the `DiagramElements` rule on line 4. In addition, the `DiagramLabels` rule defines the migration for the value of the `accessor` feature on line 26.

In Groovy-for-COPE (Listing 6.7), the migration is similar to ATL but is specified imperatively. In Listing 6.7, a loop iterates over each instance of `DiagramElement`

```
1  for (diagramElement in DiagramElement.allInstances()) {
2   diagramElement.figure = diagramElement.figure.length()
3
4   if (DiagramLabel.allInstances.contains(diagramElement)) {
5    diagramElement.accessor = diagramElement.accessor.length()
6   }
7  }
```

Listing 6.7: Simplified GMF Graph model migration in COPE

(line 1), migrating the value of its `figure` feature (line 2). The `allInstances` function is used to locate every model element with the type `DiagramElement` or one of its subtypes. If the `DiagramElement` is also a `DiagramLabel` (line 4), the value of its accessor feature is also migrated (line 5). In Groovy-for-COPE, the migration strategy uses two model operations: the assignment statements on lines 2 and 5.

In both ATL and Groovy-for-COPE, only 2 model operations are required for this migration: an assignment for each of the two features being migrated. However, the equivalent Flock migration strategy, shown in Listing 6.8, requires 5 model operations: the assignment statements on lines 2, 6, 10, 11 and 15. Note that the migration of the `figure` feature is specified four times (once for each subtype of `DiagramElement`). A single `DiagramElement` rule cannot be used to migrate the `figure` feature because, when a `migrate` rule does not specify a `to` part, Flock will create an instance of the type named after the `migrate` keyword. In other words, a `migrate DiagramElement` rule will result in Flock attempting to instantiate the abstract class `DiagramElement`. Instead migration must be specified using four migrate rules, as shown in Listing 6.8.

In the current implementation of Flock, `migrate` rules are used for specifying two concerns and the limitation described here might be avoided if those concerns were specified using two distinct language constructs. The first concern relates to,the `to` part of a `migrate` rule, which is used to establish type equivalences between the original and evolved metamodel. When a metaclass is renamed, for example, migration in Flock would typically use a rule of the form `migrate OldType to NewType`. Omitting the `to` part of a rule (`migrate X`) is a shorthand for `migrate X to X`. The second concern relates to the body of each rule, which specifies the way in which each model element should be migrated. Separating the two concerns using distinct language constructs might facilitate the re-use of migration code between subtypes. The extent to which greater re-use and increased conciseness can be addressed with changes to the implementation of Flock is discussed in Section 7.2. The sequel considers one further limitation of existing-target and conservative copy migration languages.

```
migrate Compartment {
  migrated.figure := original.figure.length();
}

migrate Connection {
  migrated.figure := original.figure.length();
}

migrate DiagramLabel {
  migrated.figure := original.figure.length();
  migrated.accessor := original.accessor.length();
}

migrate Node {
  migrated.figure := original.figure.length();
}
```

Listing 6.8: Simplified GMF Graph model migration in Flock

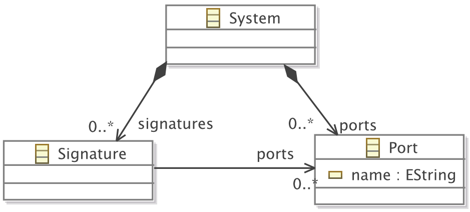
### Limitation 2: Side-effects during initialisation

The measurements observed for one of the examples of co-evolution from Chapter 4, Change Reference to Containment (Table 6.1), cannot be explained by the conceptual differences between source-target relationship. Instead, the way in which the source-target relationship is implemented must be considered.

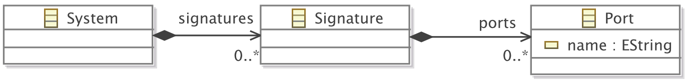
When a reference feature is changed to a containment reference during metamodel evolution, constructing the migrated model by starting from the original model (as is the case with existing-target and conservative copy) can have side-effects which complicate migration.

In the Change Reference to Containment example, a `System` initially comprises `Ports` and `Signatures` (Figure 6.13(a)). A `Signature` references any number of `ports`. The metamodel is evolved to prevent sharing of `Ports` between `Signatures` by changing the `ports` feature to a containment instead of a reference (Figure 6.13(b)). `Ports` are contained in `Signatures` rather than in `Systems`, and consequently the `ports` is no longer a feature of `System`.

Listing 6.9 shows the migration strategy using new-target in ATL. Three model operations are used: the assignment statements on lines 3, 8 and 14. The rules for migrating `Systems` (lines 1-4) and `Ports` (13-15) copy values for the features unaffected by evolution (`signatures` and `name` respectively). The rule for migrating `Signatures` (lines 6-11) clones each member of the `ports` feature (using the `Port` rule on lines 13-15). Crucially, the `Ports` rule is marked as `lazy` and consequently is only executed when called from the `Signatures` rule. By contrast, the `Systems` and `Signatures` rules are executed automatically by ATL for each `System` and `Signature` in the original model, respectively.



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.13: Change Reference to Containment metamodel evolution

```
rule Systems {
  from o : Before!System
  to m : After!System (
    signatures <- o.signatures
  )
}

rule Signatures {
  from o : Before!Signature
  to m : After!Signature (
    ports <- o.ports->collect(p | thisModule.Ports(p))
  )
}

lazy rule Ports {
  from o : Before!Port
  to m : After!Port (
    name <- o.name
  )
```

Listing 6.9: Migration for Change Reference to Containment in ATL

```
1  migrate Signature {
2    for (port in original.ports) {
3      if (migrated.ports.excludes(port.equivalent())) {
4        var clone := new Migrated!Port;
5        clone.name := port.name;
6        migrated.ports.add(clone);
7      }
8    }
9  }
10
11 delete Port when:
12   not Original!Signature.all.exists(s|s.ports.includes(original))
```

Listing 6.10: Migration for Change Reference to Containment in Flock

In existing-target and conservative copy migration languages, migration is less straightforward because, during the initialisation of the the migrated model from the original model, the value of a containment reference (`Signature#ports`) is set. When a containment reference is set, the contained objects are removed from their previous containment reference (i.e. setting a containment reference has side-effects). Therefore, in a `System` where more than one `Signature` references the same `Port`, the migrated model cannot be formed by copying the contents of `Signature#ports` from the original model. Attempting to do so causes each `Port` to be contained only in the last referencing `Signature` that was copied.

In Flock, the containment nature of the reference is enforced when the migrated model is initialised. Because changing the contents of a containment reference has side-effects, a `Port` that appears in the `ports` reference of a `Signature` in the original model may not have been automatically copied to the `ports` reference of the equivalent `Signature` in the migrated model during initialisation. Consequently, the migration strategy must check the `ports` reference of each migrated `Signature`, cloning only those `Ports` that have not be automatically copied during initialisation (see line 3 of Listing 6.10). The Flock migration strategy uses 3 model operations: assignments on lines 5 and 6, and a deletion on 11.

The Flock migration strategy must also remove any `Ports` which are not referenced by any `Signature` (line 11 of Listing 6.10), whereas the ATL migration strategy, which initialises any empty migrated model, does not copy unreferenced `Ports`.

When a non-containment reference is changed to a containment reference, migration strategies written in Flock and Groovy-for-COPE must account for the side-effects that can occur during initialisation of the migrated model, resulting in less concise migration strategies. The existing-target and conservative copy implementations used in COPE and Flock might be changed to avoid this limitation by either automatically cloning values when a reference is changed to be a containment reference, or by allowing the user to specify features that should not be copied by the source-target relationship during initialisation. Section 7.2 discusses this issue further.

### 6.2.5 Summary

By counting model operations, this section has compared, in the context of model migration, three approaches to relating source-target relationship: new-target, existing-target and conservative copy. The results have been analysed and the measurement method described.

The analysis of the measurements has shown that new- and existing-target migration languages are more concise in different situations. New-target languages require fewer model operations than existing-target languages when metamodel evolution involves the renaming of features. Existing-target languages require fewer model operations than new-target languages when metamodel evolution does not affect most model elements. For the examples considered here, the latter context was more common. Conservative copy requires fewer model operations than both new- and existing-target in almost all of the examples considered here.

The comparison has highlighted two limitations of the conservative copy algorithm implemented in Flock, and this section has shown how these limitations are problematic for specifying some types of migration strategy.

The author is not aware of any existing quantitive comparisons of migration languages, and, as such, the best practices for conducting such comparisons are not clear. The method used in obtaining these measurements has been described to provide a foundation for future comparisons.

## 6.3 Evaluating Epsilon Flock and other Co-evolution Tools

This section assesses the extent to which Epsilon Flock (Section 5.4) can be used for automating developer-driven co-evolution. To this end, Flock is compared to three co-evolution tools using an expert evaluation. While Chapter 4 highlighted theoretical differences between co-evolution tools, the expert evaluation explored the ways in which migration tools compare in practice.

One aspect of Flock, conciseness, was evaluated in Section 6.2. The evaluation performed in this section evaluates several further aspects of model migration tools. The results of the comparison, described in Section 6.3.3, suggest situations in which using Flock leads to increased productivity of model migration, and, conversely, situations in which the other co-evolution tools provide benefits over using Flock. Additionally, the expert evaluation aimed to simplify the selection of migration tools by recommending tools for particular situations or requirements. Tool selection advice was synthesised from the expert evaluation, and recommends tools that are suitable, for example, when scalability is a concern (many large models are to be migrated).

The way in which Flock affects the productivity of model migration might have been explored using a comprehensive user-study, involving hundreds of users. However, locating a large number of participants with expertise in model-driven engineering was not possible given the time constraints of the research. Alternatively, Flock and several further co-evolution tools might have been applied, by the author, to a large, independent co-evolution example in a case study. However, exploring the variations in productivity of the co-evolution tools would probably have been challenging as the author is obviously more familiar with Flock than the other tools. Instead, the comparison of co-evolution tools was performed using an expert evaluation.

The remainder of this section describes the comparison method, reports results and tool selection guidance, and discusses the situations in which Flock was identified as stronger or weaker than the other co-evolution tools. Section 6.3.1 describes the way in which the co-evolution tools were selected, comparison criteria were identified and the way in which the tools were applied to two co-evolution examples. The experts' experiences with each tool are reported in Section 6.3.2. Section 6.3.3 presents the experts' guidance for identifying the most appropriate model migration tool in different situations, and the section concludes with a description of the strengths and weaknesses of Flock.

Section 6.3 is based on joint work with Markus Herrmannsdörfer (a research student at Technische Universität München), James Williams (a research student in this department), Dimitrios Kolovos (a lecturer in this department) and Kelly Garcés (then a research student at EMN-INRIA / LINA-INRIA in Nantes), and has been published in [Rose *et al.* 2010b]. Garcés provided assistance with installing and configuration one of the migration tools, and commented on a draft of the paper. Herrmannsdörfer, Williams and Kolovos played a larger role in the comparison, as discussed in Sections 6.3.1 and 6.3.2.

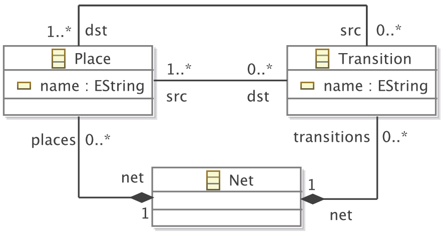
### 6.3.1 Comparison Method

The comparison described in this section is based on practical application of the tools to the co-evolution examples described below. This section also discusses the tool selection and comparison processes. Herrmannsdörfer and the author identified the co-evolution examples, and formulated the comparison process.

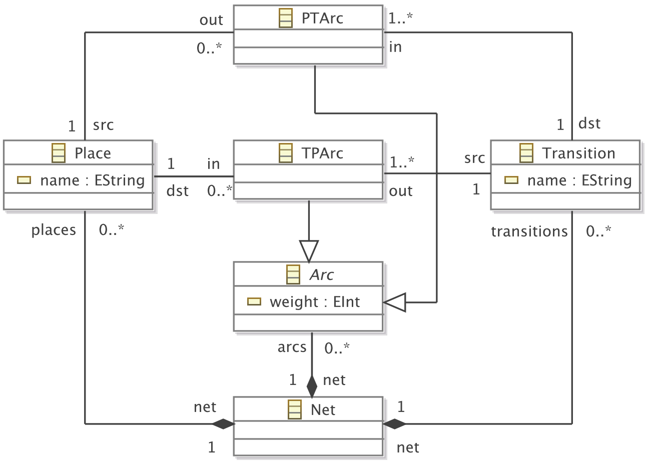
#### Co-Evolution Examples

To compare migration tools, two examples of co-evolution were used. The first, Petri nets, was first described in Section 5.3 and is a well-known problem in the model migration literature. The Petri nets example was used to test the installation and configuration of the migration tools. The second, GMF, is a larger example taken from a real-world model-driven development project, and was identified as a potentially useful example for co-evolution case studies in Chapter 4 and in [Herrmannsdoerfer *et al.* 2009b].

**Petri Nets.** The first example is the Petri nets example, which was described in Section 5.3, and is repeated in Figure 6.14. During metamodel evolution, the `PTArc` and `TPArc` classes are introduced to allow the specification of weighted arcs.



(a) Original metamodel.



(b) Evolved metamodel.

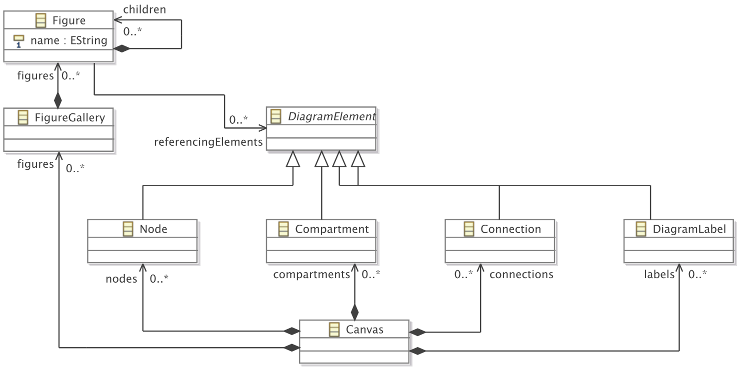
Figure 6.14: Petri nets metamodel evolution (taken from [Rose *et al.* 2010f], and from Section 5.3).

**GMF.** The second example is taken from GMF [Gronback 2009], an Eclipse project for generating graphical editors for models. The development of GMF is model-driven and utilises four domain-specific metamodels. Here, we consider one of those metamodels, GMF Graph, and its evolution between GMF versions 1.0 and 2.0. The GMF Graph example is now summarised, and more details can be found in Section C.3.1.

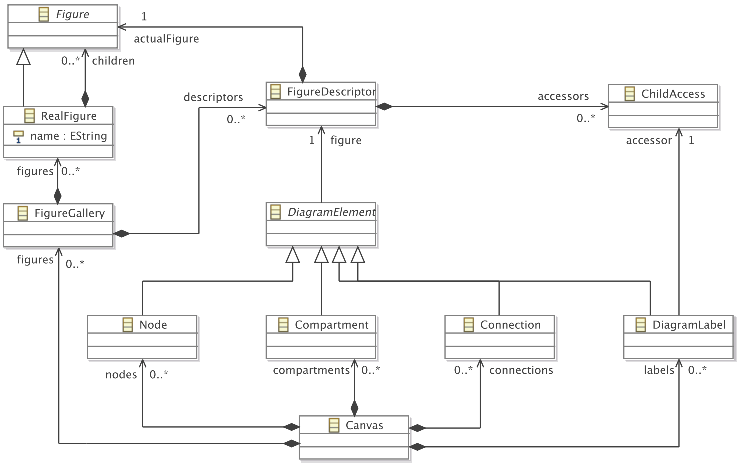
The GMF Graph metamodel (Figure 6.15) describes the appearance of the generated graphical model editor. As described in the GMF Graph documentation[3], the

---

[3] `http://wiki.eclipse.org/GMFGraph_Hints`



(a) Original metamodel.



(b) Evolved metamodel.

Figure 6.15: GMF graph metamodel evolution

Graph metamodel from GMF 1.0 was evolved – as shown in Figure 6.15(b) – to facilitate greater re-use of figures by introducing a proxy [Gamma *et al.* 1995] for `Figure`, termed `FigureDescriptor`. The original `referencingElements` reference was removed, and an extra metaclass, `ChildAccess` in its place. Section C.3.1 discusses the metamodel changes in more detail.

GMF provides a migrating algorithm that produces a model conforming to the evolved Graph metamodel from a model conforming to the original Graph metamodel. In GMF, migration is implemented using Java. The GMF source code includes two example editors, for which the source code management system contains versions conforming to GMF 1.0 and GMF 2.0. For the comparison of migration tools described in this paper, the migrating algorithm and example editors provided by GMF were used to determine the correctness of the migration strategies produced by using each model migration tool.

#### Compared Tools

The expert evaluation compares Flock (manual specification), COPE (operator-based) and AML (inference). A further tool, Ecore2Ecore (manual specification), was included because it is distributed with EMF (Section 2.3.1), arguably the most widely used modelling framework. AML, COPE and Ecore2Ecore were discussed in Chapter 4, and Flock in Chapter 5.

#### Comparison Process

The comparison of migration tools was conducted by applying each of the four tools (Ecore2Ecore, AML, COPE and Flock) to the two examples of co-evolution (Petri nets and GMF). The developers of each tool were invited to participate in the comparison. The authors of COPE and Flock were able to participate fully, while the authors of Ecore2Ecore and AML were available for guidance, advice, and to comment on preliminary results.

Each tool developer was assigned a migration tool to apply to the two co-evolution examples. Because the authors of Ecore2Ecore and AML could not participate fully in the comparison, two colleagues experienced in model transformation and migration, James Williams and Dimitrios Kolovos, took part. To improve the validity of the comparison, each tool was used by someone other than its developer.

The comparison was conducted in three phases. In the first phase, criteria against which the tools would be compared were identified by discussion between the tool developers. In the second phase, co-evolution of Petri nets was used for familiarisation with the migration tools and to assess the suitability of the comparison criteria. In the third phase, the tools were applied to the larger co-evolution (GMF) and results were drawn from the experiences of the tool developers. Table 6.3 summarises the comparison criteria used, which provide a foundation for future comparisons. The next section presents, for each criterion, observations from applying the migration tools to the co-evolution examples.

### 6.3.2 Comparison Results

This section reports the similarities and differences of each tool, using the nine criteria described above. The migration strategies formulated with each tool are available online[4].

Each subsection below considers one criterion. This section reports the experiences of the developer to which each tool was allocated. As such, this section contains the work of others. Specifically, Herrmannsdörfer described Epsilon Flock, Williams described COPE and Kolovos described Ecore2Ecore. (The author described AML, and introduced each criterion).

---

[4] `http://github.com/louismrose/migration_comparison`

| **Name** | **Description** |
|---|---|
| Construction | Ways in which tool supports the development of migration strategies |
| Change | Ways in which tool supports change to migration strategies |
| Extensibility | Extent to which user-defined extensions are supported |
| Re-use | Mechanisms for re-using migration patterns and logic |
| Conciseness | Size of migration strategies produced with tool |
| Clarity | Understandability of migration strategies produced with tool |
| Expressiveness | Extent to which migration problems can be codified with tool |
| Interoperability | Technical dependencies and procedural assumptions of tool |
| Performance | Time taken to execute migration |

Table 6.3: Summary of comparison criteria.

### Constructing the migration strategy

Facilitating the specification and execution of migration strategies is the primary function of model migration tools. This section reports the process for and challenges faced in constructing migration strategies with each tool.

**AML.** An AML user specifies a combination of match heuristics from which AML infers a migrating transformation by comparing original and evolved metamodels. Matching strategies are written in a textual syntax, which AML compiles to produce an executable workflow. The workflow is invoked to generate the migrating transformation, codified in the Atlas Transformation Language (ATL) [Jouault & Kurtev 2005]. Devising correct matching strategies is difficult, as AML lacks documentation that describes the input, output and effects of each heuristic. Papers describing AML (such as [Garcés *et al.* 2009]) discuss each heuristic, but mostly in a high-level manner. A semantically invalid combination of heuristics can cause a runtime error, while an incorrect combination results in the generation of an incorrect migration transformation. However, once a matching strategy is specified, it can be re-used for similar cases of metamodel evolution. To devise the matching strategies used in this paper, AML's author provided considerable guidance.

**COPE.** A COPE user applies *coupled operations* to the original metamodel to form the evolved metamodel. Each coupled operation specifies a metamodel evolution along with a corresponding fragment of the model migration strategy. A history of applied operations is later used to generate a complete migration strategy. As COPE is meant for co-evolution of models and metamodels, reverse engineering a large metamodel can be difficult. Determining which sequence of operations will produce a correct migration is not always straightforward. To aid the user, COPE allows operations to be undone. To help with the migration process, COPE offers the *Convergence View* which utilises EMF Compare to display the differences between two metamodels. While this was useful, it can, understandably, only provide a list of explicit differences and not the semantics of a metamodel change. Consequently, reverse-engineering a large and unfamiliar metamodel is challenging, and migration for the GMF Graph example could only be completed with considerable guidance from the author of COPE.

**Ecore2Ecore.** In Ecore2Ecore model migration is specified in two steps. In the first step, a graphical mapping editor is used to construct a model that declares basic migrations. In this step only very simple migrations such as class and feature renaming can be declared. In the next step, the developer needs to use Java to specify a customised parser (resource handler, in EMF terminology) that can parse models that conform to the original metamodel and migrate them so that they conform to the new metamodel. This customised parser exploits the basic migration information specified in the first step and delegates any changes that it cannot recognise to a particular Java method in the parser for the developer to handle. Handling such changes is tedious as the developer is only provided with the string contents of the unrecognised features and then needs to use low-level techniques – such as data-type checking and conversion, string splitting and concatenation – to address them. Here it is worth mentioning that Ecore2Ecore cannot handle all migration scenarios and is limited to cases where only a certain degree of structural change has been introduced between the original and the evolved metamodel. For cases which Ecore2Ecore cannot handle, developers need to specify a custom parser without any support for automated element copying.

**Flock.** In Flock, model migration is specified manually. Flock automatically copies only those model elements which still conform to the evolved metamodel. Hence, the user specifies migration only for model elements which no longer conform to the evolved metamodel. Due to the automatic copying algorithm, an empty Flock migration strategy always yields a model conforming to the evolved metamodel. Consequently, a user typically starts with an empty migration strategy and iteratively refines it to migrate non-conforming elements. However, there is no support to ensure that all non-conforming elements are migrated. In the GMF Graph example, completeness could only be ensured by testing with numerous models. Using this method, a migration strategy can be easily encoded for the Petri net example. For the GMF Graph example whose metamodels are larger, it was more difficult, since there is no tool support for analysing the changes between original and evolved metamodel.

### Changing the migration strategy

Migration strategies can change in at least two ways. Firstly, as a migration strategy is developed, testing might reveal errors which need to be corrected. Secondly, further metamodel changes might require changes to an existing migration strategy.

**AML.** Because AML automatically generates migrating transformations, changing the transformation, for example after discovering an error in the matching strategy, is trivial. To migrate models over several versions of a metamodel at once, the migrating transformations generated by AML can be composed by the user. AML provides no tool support for composing transformations.

**COPE.** COPE's undo feature means that incorrect migrations can be easily fixed. COPE stores a history of *releases* – a set of operations that has been applied between versions of the metamodel. Because the migration code generated from the release history can migrate models conforming to any previous metamodel release, COPE provides a comprehensive means for chaining migration strategies.

**Ecore2Ecore.** Migrations specified using Ecore2Ecore can be modified via the graphical mapping editor and the Java code in the custom model parser. Therefore, developers can use the features of the Eclipse Java IDE to modify and debug migrations. Ecore2Ecore provides no dedicated tool support for composing migrations.

**Flock.** There is comprehensive support in Flock for fixing errors. A migration strategy can easily be re-executed using a launch configuration, and migration errors are linked to the line in the migration strategy that caused the error to occur. If the metamodel is further evolved, the original migration strategy has to be extended, since there is no explicit support to chain migration strategies. The full migration strategy may need to be read to know where to extend it.

### Extensibility

The fundamental constructs used for specifying migration in COPE and AML (operators and match heuristics, respectively) are extensible. Flock and Ecore2Ecore use a more imperative (than declarative) approach, and as such do not provide extensible constructs.

**AML.** An AML user can specify additional matching heuristics. This requires understanding of AML's domain-specific language for manipulating the data structures from which migrating transformations are generated.

**COPE** provides the user with a large number of operations. If there is no applicable operation, a COPE user can write their own operations using an in-place transformation language embedded into Groovy[5].

---

[5] `http://groovy.codehaus.org/`

### Re-use

Each migration tool captures patterns that commonly occur in model migration. This section considers the extent to which the patterns captured by each tool facilitate re-use between migration strategies.

**AML.** Once a matching strategy is specified, it can potentially be re-used for further cases of metamodel evolution. Match heuristics provide a re-usable and extensible mechanism for capturing metamodel change and model migration patterns.

**COPE.** An operation in COPE represents a commonly occurring pattern in metamodel migration. Each operation captures the metamodel evolution and model migration steps. Custom operations can be written and re-used.

**Ecore2Ecore.** Mapping models cannot be reused or extended in Ecore2Ecore but as the custom model parser is specified in Java, developers can decompose it into reusable parts some of which can potentially be reused in other migrations.

**Flock.** A migration strategy encoded in Flock is modularised according to the classes whose instances need migration. There is support to reuse code within a strategy by means of operations with parameters and across strategies by means of imports. Re-use in Flock captures only migration patterns, and not the higher level co-evolution patterns captured in COPE or AML.

### Conciseness

A concise migration strategy is arguably more readable and requires less effort to write than a verbose migration strategy. This section comments on the conciseness of migration strategies produced with each tool, and reports the lines of code (without comments and blank lines) used.

**AML.** 117 lines were automatically generated for the Petri nets example. 563 lines were automatically generated for the GMF Graph example, and a further 63 lines of code were added by hand to complete the transformation. Approximately 10 lines of the user-defined code could be removed by restructuring the generated transformation.

**COPE** requires the user to apply operations. Each operation application generates one line of code. The user may also write additional migration code. For the Petri net example, 11 operations were required to create the migrator and no additional code. The author of COPE migrated the GMF Graph example using 76 operations and 73 lines of additional code.

**Ecore2Ecore.** As discussed above, handling changes that cannot be declared in the mapping model is a tedious task and involves a significant amount of low level code. For the PetriNets example, the Ecore2Ecore solution involved a mapping model containing 57 lines of (automatically generated) XMI and a custom hand-written resource handler containing 78 lines of Java code.

**Flock.** 16 lines of code were necessary to encode the Petri nets example, and 140 lines of code were necessary to encode the GMF Graph example. In the GMF Graph example, approximately 60 lines of code implement missing built-in support for rule inheritance, even after duplication was removed by extracting and re-using a subroutine.

### Clarity

Because migration strategies can change and might serve as documentation for the history of a metamodel, their clarity is important. This section reports on aspects of each tool that might affect the clarity of migration strategies.

**AML.** The AML code generator takes a conservative approach to naming variables, to minimise the chances of duplicate variable names. Hence, some of the generated code can be difficult to read and hard to re-use if the generated transformation has to be completed by hand. When a complete transformation can be generated by AML, clarity is not as important.

**COPE.** Migration strategies in COPE are defined as a sequence of operations. The release history stores the set of operations that have been applied, so the user is clearly able to see the changes they have made, and find where any issues may have been introduced.

**Ecore2Ecore.** The graphical mapping editor provided by Ecore2Ecore allows developers to have a high-level visual overview of the simple mappings involved in the migration. However, migrations expressed in the Java part of the solution can be far more obscure and difficult to understand as they mix high-level intention with low-level string management operations.

**Flock** clearly states the migration strategy from the source to the target metamodel. However, the boilerplate code necessary to implement rule inheritance slightly obfuscates the real migration code.

### Expressiveness

Migration strategies are easier to infer for some categories of metamodel change than others [Gruschko *et al.* 2007]. This section reports on the ability of each tool to migrate the examples considered in this comparison.

**AML.** A complete migrating transformation could be generated for the Petri nets example, but not for the GMF Graph example. The latter contains examples of two complex changes that AML does not currently support[6]. Successfully expressing the GMF Graph example in AML would require changes to at least one of AML's heuristics. However, AML provided an initial migration transformation that was completed by hand.

In general, AML cannot be used to generate complete migration strategies for co-evolution examples that contain *breaking and non-resolvable changes*, according to the categorisations discussed in Section 3.2.4.

**COPE.** The expressiveness of COPE is defined by the set of operations available. The Petri net example was migrated using only built-in operations. The GMF Graph example was migrated using 76 built-in operations and 2 user-defined migration actions. Custom migration actions allow users to specify any migration strategy.

**Ecore2Ecore.** A complete migration strategy could be generated for the Petri nets example, but not for the GMF Graph example. The developers of Ecore2Ecore have advised that the latter involves significant structural changes between the two versions and recommended implementing a custom model parser from scratch.

**Flock.** Since Flock extends EOL, it is expressive enough to encode both examples. However, Flock does not provide an explicit construct to copy model elements and thus it was necessary to call Java code from within Flock for the GMF Graph example.

### Interoperability

Migration occurs in a variety of settings with differing requirements. This section considers the technical dependencies and procedural assumptions of each tool, and seeks to answer questions such as: "Which modelling technologies can be used?" and "What assumptions does the tool make on the migration process?"

**AML** depends only on ATL, while its development tools also require Eclipse. AML assumes that the original and target metamodels are available for comparison, and does not require a record of metamodel changes. AML can be used with either Ecore (EMF) or KM3 metamodels.

**COPE** depends on EMF and Groovy, while its development tools also require Eclipse and EMF Compare. COPE does not require both the original and target metamodels to be available. When COPE is used to create a migration strategy after metamodel evolution has already occurred, the metamodel changes must be reverse-engineered. To facilitate this, the target metamodel can be used with the Convergence View, as

---

[6] `http://www.eclipse.org/forums/index.php?t=rview&goto=526894`

discussed in Section 6.3.2. COPE targets EMF, and does not support other modelling technologies.

**Ecore2Ecore** depends only on EMF. Both the original and the evolved versions of the metamodel are required to specify the mapping model with the Ecore2Ecore development tools. Alternatively, the Ecore2Ecore mapping model can be constructed programmatically and without using the original metamodel[7]. Unlike the other tools considered, Ecore2Ecore does not require the original metamodel to be available in the workspace of the metamodel user.

**Flock** depends on Epsilon and its development tools also require Eclipse. Flock assumes that the original and target metamodels are available for encoding the migration strategy, and does not require a record of metamodel changes. Flock can be be used to migrate models represented in EMF, MDR, XML and Z (CZT), although we only encoded a migration strategy for EMF metamodels in the presented examples.

### Performance

The time taken to execute model migration is important, particularly once a migration strategy has been distributed to metamodel users. Ideally, migration tools will produce migration strategies whose execution time is quick and scales well with large models.

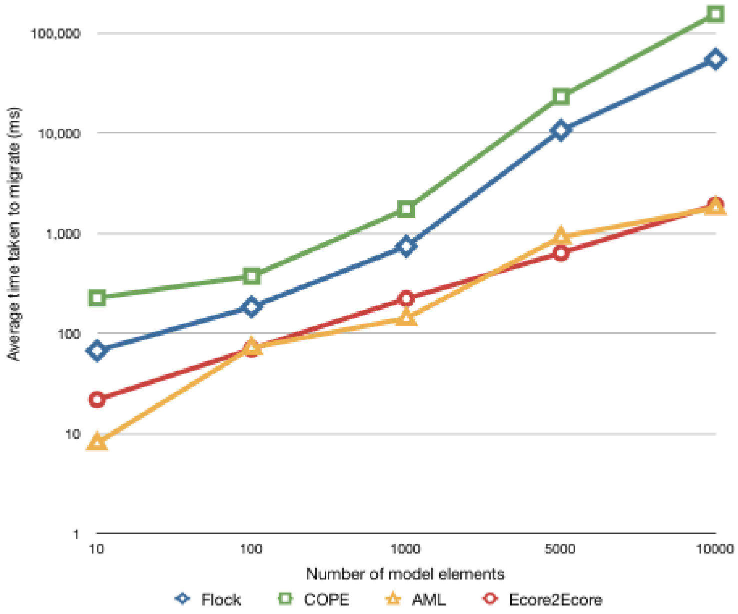


Figure 6.16: Migration tool performance comparison.

---

[7] Private communication with Marcelo Paternostro, an Ecore2Ecore developers.

To measure performance, five sets of Petri net models were generated at random. Models in each set contained 10, 100, 1000, 5,000, and 10,000 model elements. Figure 6.16 shows the average time taken by each tool to execute migration across 10 repetitions for models of different sizes. Note that the Y axis has a logarithmic scale. The results indicate that, for the Petri nets co-evolution example, AML and Ecore2Ecore execute migration significantly more quickly than COPE and Flock, particularly when the model to be migrated contains more than 1,000 model elements. Figure 6.16 indicates that, for the Petri nets co-evolution example, Flock executes migration between two and three times faster than COPE, although the author of COPE reports that turning off validation causes COPE to perform similarly to Flock.

### 6.3.3 Discussion

The comparison described above highlights similarities and differences between a representative sample of model migration approaches. From this comparison, guidance for selecting between tools was synthesised. The guidance is presented below, and was produced by all four participants in the comparison (Herrmannsdörfer, Williams, Kolovos and the author).

COPE captures co-evolution patterns (which apply to both model and metamodel), while Ecore2Ecore, AML and Flock capture only model migration patterns (which apply just to models). Because of this, COPE facilitates a greater degree of re-use in model migration than other approaches. However, the order in which the user applies patterns with COPE impacts on both metamodel evolution and model migration, which can complicate pattern selection particularly when a large amount of evolution occurs at once. The re-usable co-evolution patterns in COPE make it well suited to migration problems in which metamodel evolution is frequent and in small steps.

Flock, AML and Ecore2Ecore are preferable to COPE when metamodel evolution has occurred before the selection of a migration approach. Because of its use of co-evolution patterns, we conclude that COPE is better suited to forward- rather than reverse-engineering.

Through its Convergence View and integration with the EMF metamodel editor, COPE facilitates metamodel analysis that is not possible with the other approaches considered in this paper. COPE is well-suited to situations in which measuring and reasoning about co-evolution is important.

In situations where migration involves modelling technologies other than EMF, AML and Flock are preferable to COPE and Ecore2Ecore. AML can be used with models represented in KM3, while Flock can be used with models represented in MDR, XML and CZT. Via the connectivity layer of Epsilon, Flock can be extended to support further modelling technologies.

There are situations in which Ecore2Ecore or AML might be preferable to Flock and COPE. For large models, Ecore2Ecore and AML might execute migration significantly more quickly than Flock and COPE. Ecore2Ecore is the only tool that has no technical dependencies (other than a modelling framework). In situations where migration must be embedded in another tool, Ecore2Ecore offers a smaller footprint than other migration approaches. Compared to the other approaches considered in this paper, AML automatically generates migration strategies with the least guidance from the user.

Despite these advantages, Ecore2Ecore and AML are unsuitable for some types of migration problem, because they are less expressive than Flock and COPE. Specifically, changes to the containment of model elements typically cannot be expressed with Ecore2Ecore, and *metamodel-specific changes* [Herrmannsdoerfer *et al.* 2008a] cannot be expressed with AML. Because of this, it is important to investigate metamodel changes before selecting a migration tool. Furthermore, it might be necessary to anticipate which types of metamodel change are likely to arise before selecting a migration tool. Investing in one tool to discover later that it is no longer suitable causes wasted effort.

| **Requirement** | **Recommended Tools** |
|---|---|
| Frequent, incremental co-evolution | COPE |
| Reverse-engineering | AML, Ecore2Ecore, Flock |
| Modelling technology diversity | Flock |
| Quicker migration for larger models | AML, Ecore2 Ecore |
| Minimal dependencies | Ecore2Ecore |
| Minimal hand-written code | AML, COPE |
| Minimal guidance from user | AML |
| Support for metamodel-specific migrations | COPE, Flock |

Table 6.4: Summary of tool selection advice. (Tools are ordered alphabetically).

### Strengths and Weaknesses of Flock

The comparison and guidance highlight strengths and weaknesses of AML, COPE, Ecore2Ecore and Flock. The findings for Flock are now summarised.

**Strengths** Flock was the only co-evolution tool suitable for performing model migration when the original and evolved metamodels are specified in different modelling technologies. AML, Ecore2Ecore and COPE are interoperable with a single modelling technology, EMF. Migrating models between metamodels represented in different modelling technologies would require modification of the co-evolution tool when using AML, Ecore2Ecore or COPE and hence, model migration with Flock requires less effort than using AML, Ecore2Ecore or COPE when migrating between modelling technologies. This was a key requirement for the co-evolution example described in the sequel.

For the examples of metamodel evolution explored here, Flock (and COPE) is more expressive than AML, but requires more guidance from the user. This is consistent with the trade-off between flexibility and level of automation of co-evolution approaches identified in Chapter 4.

Unlike COPE, Flock (and AML and Ecore2Ecore) does not make assumptions on the way in which metamodel evolution will be specified. With Flock, AML and

Ecore2Ecore, metamodel evolution need not occur at the same time or in the same development environment as the formulation of the model migration strategy. For this reason, Flock (and AML and Ecore2Ecore) arguably lead to more productive model migration when used to formulate a model migration strategy after metamodel evolution has already been specified, as was the case for the GMF Graph example used in this section.

**Weaknesses** The results presented here indicate that model migration with Flock takes longer to execute than with AML and Ecore2Ecore. This is likely because Flock migration strategies are interpreted, while AML and Ecore2Ecore migration strategies are compiled. A compiler for Flock would be likely to increase execution time, but, at present, Epsilon (which Flock extends and reuses ) lacks the infrastructure required for constructing compilers. As such, model migration with Flock is likely to be less productive than with AML or Ecore2Ecore when a large model or a large number of models are to be migrated.

Compared to COPE and AML, Flock lacks re-use of model migration patterns across varying metamodels. In Flock, model migration is specified in terms of concrete metamodel types and cannot be re-used for different metamodels. By contrast, COPE and AML capture model migration in a metamodel-independent manner. When migration is likely to be a commonly occurring practice, the use of COPE or AML rather than Flock is likely to led to increased productivity of model migration, because the metamodel-independent migration patterns will probably increase re-use and provide a vocabulary for describing migration. Section 7.2 describes ways in which Flock might be extended to capture metamodel-independent migration patterns.

### 6.3.4 Summary

The work presented in this section compared a representative sample of approaches to automating developer-driven co-evolution using an expert evaluation. The comparison was performed by following a methodical process and using an example from a real-world MDE project. Some preliminary recommendations and guidelines in choosing a co-evolution tool were synthesised from the presented results and are summarised in Table 6.4. The comparison was carried out by the tool developers (or stand-ins where the developers were unable to participate fully). Each developer used a tool other than their own so that the comparison could more closely emulate the level of expertise of a typical user.

The results of the comparison suggested situations in which the use of Flock might lead to increased productivity of model migration, and, conversely, situations in which an alternative tool might be preferable. The comparison results suggest that Flock is well-suited to co-evolution when models are to be migrated between different modelling technologies, when migration involves metamodel-specific detail, and when metamodel evolution has occurred prior to – or in a different development environment to – the formulation of a model migration strategy. Additionally, Flock might be improved via optimisations to increase execution speed for large models or a large number of models, and by considering the ways in which model migration patterns could be captured in a metamodel-independent manner.

Some criteria were excluded from the comparison because of the method employed. For instance, the learnability of a tool affects the productivity of users, and, as such, affects tool selection. However, drawing conclusions about learnability (and also usability) is challenging with the comparison method employed because of the subjective nature of these characteristics. A comprehensive user study (with hundreds of users) would be more suitable for assessing these types of criteria.

## 6.4 Evaluating Co-Evolution Tools with an Example from UML

In contrast to the previous section, which compared Flock to three co-evolution tools, the evaluation performed in this section compares Flock with model-to-model transformation tools. As discussed in Chapter 4, model migration can be regarded as a specialisation of model-to-model transformation. Chapter 5 introduces Flock, a language tailored for model migration. This section compares Flock with other model-to-model transformation languages, and explores the benefits and drawbacks of treating model migration and model-to-model transformation as separate model management operations, as proposed in this thesis and by [Sprinkle 2003].

The author participated in the 2010 edition of the Transformation Tools Contest (TTC), a workshop series that seeks to compare and contrast tools for performing model and graph transformation. At TTC 2010[8], two rounds of submissions were invited: cases (transformation problems, three of which are selected by the workshop organisers) and solutions to the selected cases. The author submitted a case based on a model migration problem from a real-world example of metamodel evolution. Nine solutions were submitted for the case, including one by the author, which used Flock.

Compared to the evaluation described in Section 6.3, the evaluation in this section compares Flock to a wider range of tools (model and graph transformation tools, and not just model migration tools). The remainder of this section describes the model migration case (Section 6.4.1), the Flock solution (Section 6.4.2), and reports the results of the workshop in which the solutions were compared and scored by the organisers and participants.

### 6.4.1 Model Migration Case

To compare Flock with other transformation tools for specifying model migration, the author submitted a case to TTC based on the evolution of the UML. The way in which activity diagrams are modelled in the UML changed significantly between versions 1.4 and 2.1 of the specification. In the former, activities were defined as a special case of

---

[8] `http://www.planet-research20.org/ttc2010/index.php?Itemid=132`

state machines, while in the latter they are defined with a more general semantic base[9] [Selic 2005].

The remainder of this section briefly introduces UML activity diagrams, describes their evolution, and discusses the way in which solutions were assessed. Section C.2.1 describes the metamodel evolution in more detail. The work presented in this section is based on the case submitted to TTC 2010 [Rose *et al.* 2010e].

### Activity Diagrams in UML

Activity diagrams are used for modelling lower-level behaviours, emphasising sequencing and co-ordination conditions. They are used to model business processes and logic [OMG 2007b]. Figure 6.17 shows an activity diagram for filling orders. The diagram is partitioned into three *swimlanes*, representing different organisational units. *Activities* are represented with rounded rectangles and *transitions* with directed arrows. *Fork* and *join* nodes are specified using a solid black rectangle. *Decision* nodes are represented with a diamond. Guards on transitions are specified using square brackets. For example, in Figure 6.17 the transition to the restock activity is guarded by the condition `[not in stock]`. Text on transitions that is not enclosed in square brackets represents a trigger event. In Figure 6.17, the transition from the restock activity occurs on receipt of the asynchronous signal called `receive stock`. Finally, the transitions between activities might involve interaction with objects. In Figure 6.17, the Fill Order activity leads to an interaction with an object called `Filled Object`.

Between versions 1.4 and 2.2 of the UML specification, the metamodel for activity diagrams has changed significantly. The sequel summarises most of the changes, and further details can be found in the UML 1.4 [OMG 2001] andUML 2.2 [OMG 2007b] specifications.

### Evolution of Activity Diagrams

Figures 6.18 and 6.19 are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures 6.18 and 6.19.

Some differences between Figures 6.18 and 6.19 are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

To facilitate the comparison of solutions, the model shown in Figure 6.17 was used. Solutions migrated the activity diagram shown in Figure 6.17 – which conforms to UML 1.4 – to conform to UML 2.2. The UML 1.4 model, the migrated UML 2.2 model, and the UML 1.4 and 2.2 metamodels are available from[10].

Submissions were evaluated using the following criteria, which were decided in advance by the author and the workshop organisers:

---

[9] A variant of generalised coloured Petri nets.

[10] `http://www.cs.york.ac.uk/~louis/ttc/`

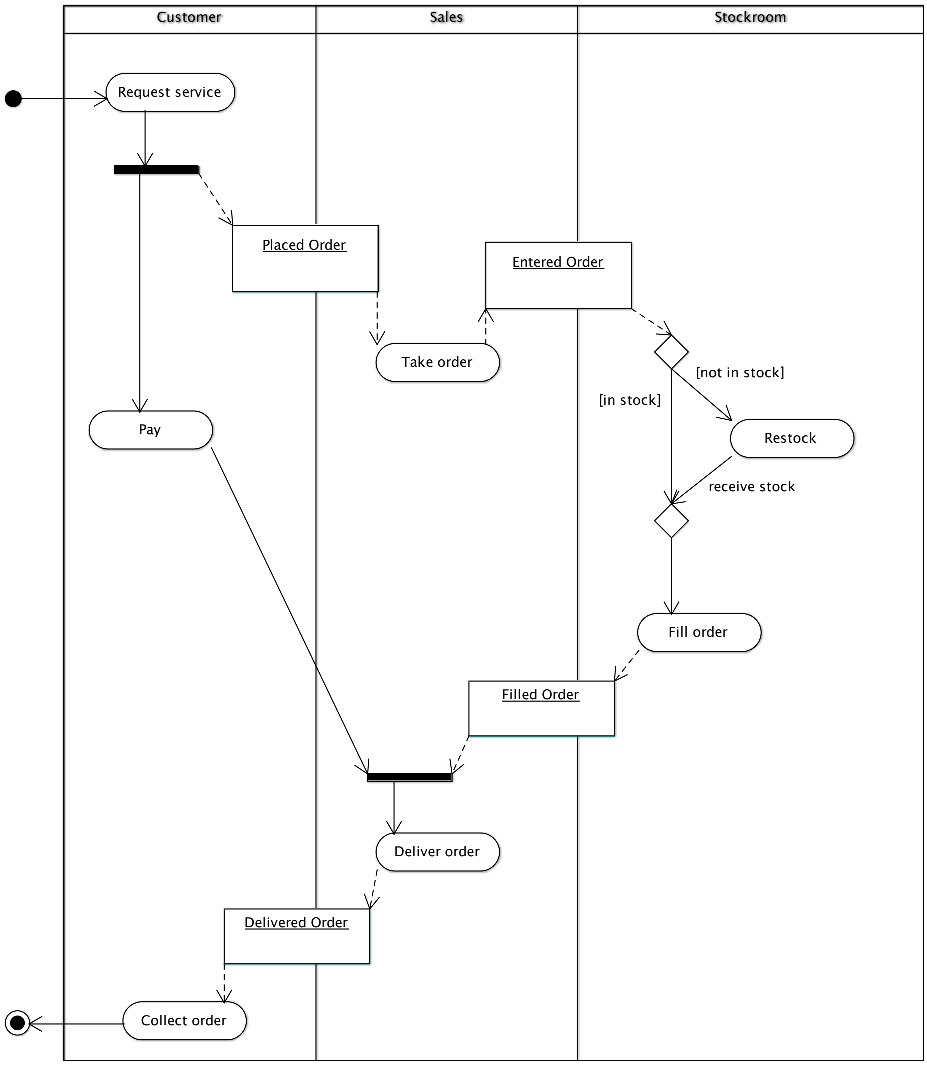


Figure 6.17: Activity model in UML 1.4, taken from [Rose *et al.* 2010e] and based on [OMG 2001, pg3-165].

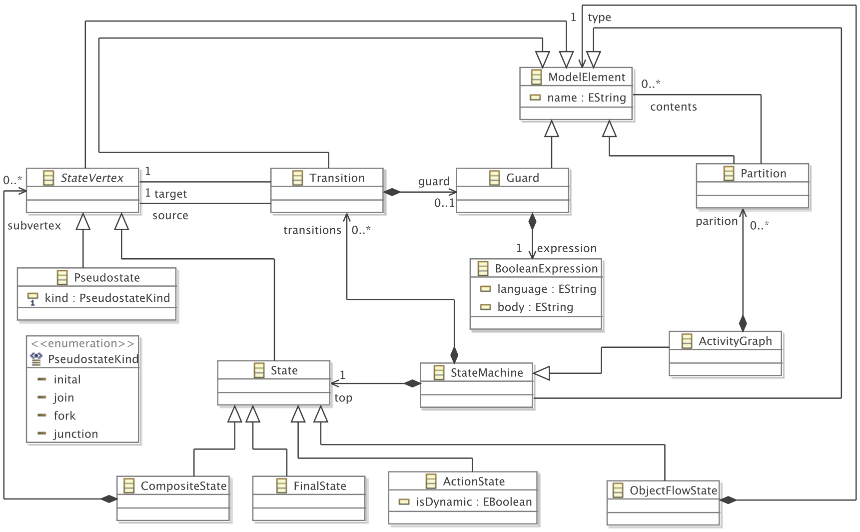


Figure 6.18: UML 1.4 Activity Graphs (based on [OMG 2001]).

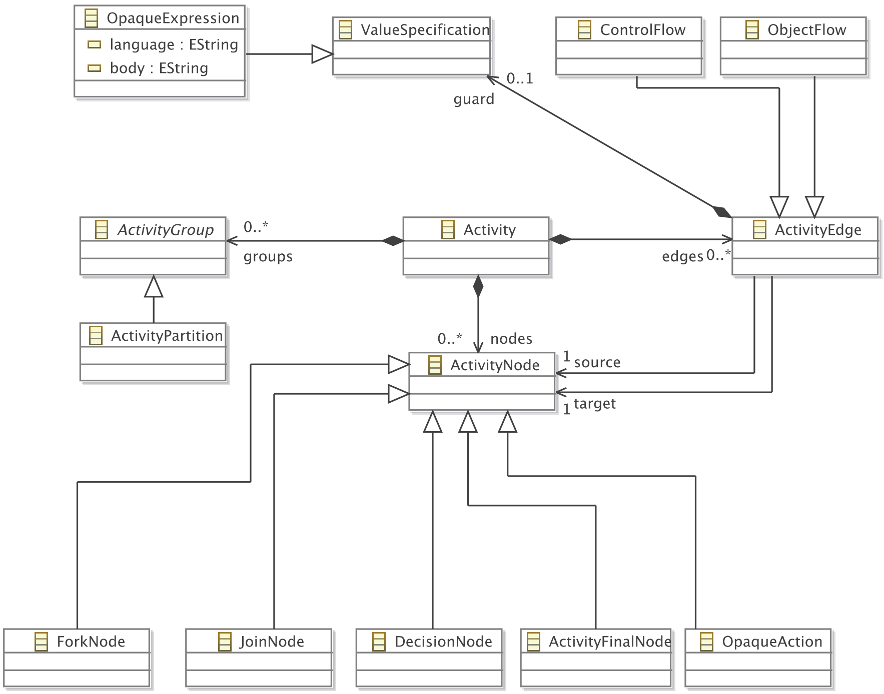


Figure 6.19: UML 2.2 Activity Diagrams (based on [OMG 2007b]).

- **Correctness**: Does the transformation produce a model equivalent to the migrated UML 2.2. model included in the case resources?
- **Conciseness**: How much code is required to specify the transformation? (Sprinkle and Karsai propose that the amount of effort required to codify migration should be directly proportional to the number of changes between original and evolved metamodel [Sprinkle & Karsai 2004]).
- **Clarity**: How easy is it to read and understand the used transformation? (For example, is a well-known or standardised language?)
- **Appropriateness**: How much effort is required to adapt the tool in providing a solution?
- **Tool maturity**: To what extent can the tool be used by people other than the developer?
- **Reproducibility**: Can the solution be reproduced on another machine?[11]
- **Extensions**: Which of the case extensions (described below) were implemented in the solution?

To further distinguish between solutions, three extensions to the core task were proposed. The first extension was added after the case was submitted, and was proposed by the workshop organisers and the solution authors. The second and third extension were included in the case by the author.

### Extension 1: Alternative Object Flow State Migration Semantics

Following the submission of the case to the competition, discussion on the TTC forums[12] revealed an ambiguity in the UML 2.2 specification indicating that the migration semantics for the ObjectFlowState UML 1.4 concept are not clear from the UML 2.2 specification. The case was revised to incorporate both the original semantics (suggested by the author and described above) and an alternative semantics (suggested by a workshop participant via the TTC forums) for migrating `ObjectFlowStates`. The alternative semantics are now described.

**In the core task** described above, instances of `ObjectFlowState` were migrated to instances of `ObjectNode`. Any instances of `Transition` that had an `ObjectFlowState` as their source or target were migrated to instances of `ObjectFlow`. Figure 6.20 shows an example application of this migration semantics. Structures such as the one shown in Figure 6.20(a) are migrated to an equivalent structure shown in

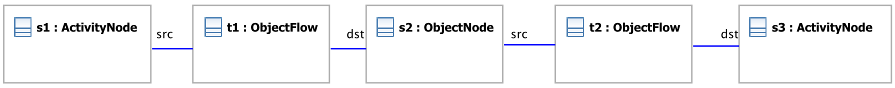
---

[11] Participants were invited to install their tools and solutions on virtual machines, which would later be made accessible via the workshop proceedings.

[12] `http://planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewgroup&groupid=4&Itemid=150` (registration required)

Figure 6.20(b). The `Transitions`, `t1` and `t2`, are migrated to instances of `ObjectFlow`. Likewise, the instance of `ObjectFlowState`, `s2`, is migrated to an instance of `ObjectNode`.

(a) ObjectFlowState structure in UML 1.4



(b) Equivalent ObjectNode structure in UML 2.2

Figure 6.20: Migrating Actions for the Core Task

**This extension** considered an alternative migration semantics for ObjectFlowState. For this extension, instances of `ObjectFlowState` (and any connected `Transitions`) were migrated to instances of `ObjectFlow`, as shown in Figure 6.21 in which the UML 2.2 `ObjectFlow`, `f1`, replaces `t1`, `t2` and `s2`.

(a) ObjectFlowState structure in UML 1.4

(b) Equivalent ObjectFlow structure in UML 2.2

Figure 6.21: Migrating Actions for Extension 1

### Extension 2: Concrete Syntax

The second extension relates to the appearance of activity diagrams. The UML specifications provide no formally defined metamodel for the concrete syntax of UML diagrams. However, some UML tools store diagrammatic information in a structured manner using XML or a modelling tool. For example, the Eclipse UML 2 tools[13] store diagrams as GMF [Gronback 2009] diagram models.

Submissions were invited to explore the feasibility of migrating the concrete syntax of the activity diagram shown in Figure 6.17 to the concrete syntax in their chosen UML 2 tool. To facilitate this, the case resources included an ArgoUML project[14] containing the activity diagram shown in Figure 6.17.

### Extension 3: XMI

The UML specifications [OMG 2001, OMG 2007b] indicate that UML models should be stored using XMI. However, because XMI has evolved at the same time as UML, UML 1.4 tools most likely produce XMI of a different version to UML 2.2 tools. For instance, ArgoUML produces XMI 1.2 for UML 1.4 models, while the Eclipse UML2 tools produce XMI 2.1 for UML 2.2.

As an extension to the core task, submissions were invited to consider how to migrate a UML 1.4 model represented in XMI 1.x to a UML 2.1 model represented in XMI 2.x. To facilitate this, the UML 1.4 model shown in Figure 6.17 was made available in XMI 1.2 as part of the case resources.

Following the submission of the case, Tom Morris, the project leader for ArgoEclipse and a committer on ArgoUML, encouraged solutions to consider the extension described above. ArgoUML cannot, at present, migrate models from UML 1 to UML 2. On the TTC forums, Morris stated that "We have nothing available to fill this hole currently, so any contributions would be hugely valuable. Not only would achieve academic fame and glory from the contest, but you'd get to see your code benefit users of one of the oldest (10+ yrs) open source UML modeling tools." [15]

## 6.4.2 Model Migration Solution in Epsilon Flock

This section describes a Flock solution for migrating UML activity diagrams in response to the evolution described above. The solution was developed by the author, and, at the workshop, compared with migration strategies written in other languages. The workshop participants and organisers rated each tool.

The Flock migration strategy was developed in an iterative and incremental manner, using the following process, starting with an empty migration strategy:

1. Execute Flock on the original model, producing a migrated model.

---

[13] `http://www.eclipse.org/modeling/mdt/?project=uml2tools`

[14] `http://argouml.tigris.org/`

[15] `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required)

2. Compare the migrated model with the reference model provided in the case resources.
3. Change the Flock migration strategy.
4. Repeat until the migrated and reference models were the same.

The remainder of this section presents the Flock solution in an incremental manner. The code listings in this section show only those rules relevant to the iteration being discussed.

### Actions, Transitions and Final States

Development of the migration strategy began by executing an empty Flock migration strategy on the original model. Because Flock automatically copies model elements that have not been affected by evolution, the resulting model contained `Pseudostates` and `Transitions`, but none of the `ActionStates` from the original model. In UML 2.2 activities, `OpaqueActions` replace `ActionStates`. Listing 6.11 shows the Flock code for changing `ActionStates` to corresponding `OpaqueActions`.

```
migrate ActionState to OpaqueAction
```

Listing 6.11: Migrating Actions

Next, similar rules were added to migrate instances of `FinalState` to instances of `ActivityFinalNode` and to migrate instances of `Transition` to `ControlFlow`, as shown in Listing 6.12.

```
migrate FinalState to ActivityFinalNode
migrate Transition to ControlFlow
```

Listing 6.12: Migrating FinalStates and Transitions

### Pseudostates

Development continued by selecting further types of state that were not present in the migrated model, such as `Pseudostatess`, which are not used in UML 2.2 activities. Instead, UML 2.2 activities use specialised `Nodes`, such as `InitialNode`. Listing 6.13 shows the Flock code used to change `Pseudostates` to corresponding `Nodes`.

```
migrate Pseudostate to InitialNode when: original.kind = Original!
    PseudostateKind#initial
migrate Pseudostate to DecisionNode when: original.kind = Original!
    PseudostateKind#junction
migrate Pseudostate to ForkNode when: original.kind = Original!PseudostateKind#
    fork
migrate Pseudostate to JoinNode when: original.kind = Original!PseudostateKind#
    join
```

Listing 6.13: Migrating Pseudostates

### Activities

In UML 2.2, `Activitys` no longer inherit from state machines. As such, some of the features defined by `Activity` have been renamed. Specifically, `transitions` has become `edges` and `partitions` has become `groups`. Furthermore, the states (or nodes in UML 2.2 parlance) of an `Activity` are now contained in a feature called `nodes`, rather than in the `subvertex` feature of a composite state accessed via the `top` feature of `Activity`. The Flock migration rule shown in Listing 6.14 captured these changes.

```
migrate ActivityGraph to Activity {
  migrated.edge = original.transitions.equivalent();
  migrated.group = original.partition.equivalent();
  migrated.node = original.top.subvertex.equivalent();
}
```

Listing 6.14: Migrating ActivityGraphs

Note that the rule in Listing 6.14 used the built-in `equivalent` operation to find migrated model elements from original model elements. As discussed in Section 5.4, the `equivalent` operation invokes other migration rules where necessary and caches results to improve performance.

Next, a similar rule for migrating `Guards` was added. In UML 1.4, the the `guard` feature of `Transition` references a `Guard`, which in turn references an `Expression` via its `expression` feature. In UML 2.2, the `guard` feature of `Transition` references an `OpaqueExpression` directly. Listing 6.15 captures this in Flock.

```
migrate Guard to OpaqueExpression {
  migrated.body.add(original.expression.body);
}
```

Listing 6.15: Migrating Guards

### Partitions

In UML 1.4 activity diagrams, `Partition` specifies a single containment reference for its `contents`. In UML 2.2 activity diagrams, partitions have been renamed to `ActivityPartitions` and specify two containment features for their contents, `edges` and `nodes`. Listing 6.16 shows the rule used to migrate `Partitions` to `ActivityPartitions` in Flock. The body of the rule shown in Listing 6.16 uses the *collect* operation to segregate the `contents` feature of the original model element into two parts.

```
migrate Partition to ActivityPartition {
  migrated.edges = original.contents.collect(e:Transition | e.equivalent());
  migrated.nodes = original.contents.collect(n:StateVertex | n.equivalent());
}
```

Listing 6.16: Migrating Partitions

### ObjectFlows

Finally, two rules were written for migrating model elements relating to object flows. In UML 1.4 activity diagrams, object flows are specified using `ObjectFlowState`, a subtype of `StateVertex`. In UML 2.2 activity diagrams, object flows are modelled using a subtype of `ObjectNode`. In UML 2.2 flows that connect to and from `ObjectNodes` must be represented with `ObjectFlows` rather than `ControlFlows`.

Listing 6.17 shows the Flock rule used to migrate `Transitions` to `ObjectFlows`. The rule applies for `Transitions` whose source or target `StateVertex` is of type `ObjectFlowState`.

```
migrate ObjectFlowState to ActivityParameterNode

migrate Transition to ObjectFlow when: original.source.isTypeOf(ObjectFlowState)
      or original.target.isTypeOf(ObjectFlowState)
```

Listing 6.17: Migrating ObjectFlows

In addition to the core task, the Flock solution also approached two of the three extensions described in the case (Section 6.4.1). The solutions to the extensions are now discussed.

### Alternative ObjectFlowState Migration Semantics

The first extension required submissions to consider an alternative migration semantics for `ObjectFlowState`, in which a single `ObjectFlow` replaces each `ObjectFlowState` and any connected `Transitions`.

Listing 6.18 shows the Flock source code used to migrate `ObjectFlowStates` (and connecting `Transitions`) to a single `ObjectFlow`. This rule was used instead of the two rules defined in Listing 6.17. In the body of the rule shown in Listing 6.18, the `source` of the `Transition` is copied directly to the `source` of the `ObjectFlow`. The `target` of the `ObjectFlow` is set to the target of the first outgoing `Transition` from the `ObjectFlowState`.

```
migrate Transition to ObjectFlow when: original.target.isTypeOf(ObjectFlowState)
      {
  migrated.source = original.source.equivalent();
  migrated.target = original.target.outgoing.first.target.equivalent();
}
```

Listing 6.18: Migrating ObjectFlowStates to a single ObjectFlow

Because, in this alternative semantics, `ObjectFlowStates` are represented as edges rather than nodes, the partition migration rule was changed such that `ObjectFlowStates` were not copied to the `nodes` feature of `Partitions`. To filter out the `ObjectFlowStates`, line 3 of Listing 6.16 was changed to include a reject statement, as shown on line 3 of Listing 6.19.

```
1 migrate Partition to ActivityPartition {
```

```
  migrated.edges = original.contents.collect(e:Transition | e.equivalent());
  migrated.nodes = original.contents.reject(ofs:ObjectFlowState | true).collect(n:Original!StateVertex | n.equivalent());
}
```

Listing 6.19: Migrating Partitions without ObjectFlowStates

The complete source code listing for the Flock migration strategy is provided in Section C.2.1.

### XMI

The second extension required submissions to migrate an activity graph conforming to UML 1.4 and encoded in XMI 1.2 to an equivalent activity graph conforming to UML 2.2 and encoded in XMI 2.1. The core task did not require submissions to consider changes to XMI (the model storage representation), but, in practice, this is a challenge to migration, as noted by Tom Morris on the TTC forums[16].

As discussed in Section 5.4, Flock extends and reuses Epsilon, which includes a model connectivity layer (EMC). EMC provides a common interface for accessing and persisting models. Currently, EMC supports EMF (XMI 2.x), MDR (XMI 1.x), and plain XML models. To support migration between metamodels defined in heterogeneous modelling frameworks, EMC was extended during the development of Flock to provide a conformance checking service.

Consequently, the migration strategy developed for the core task works for all of the types of model supported by EMC. To migrate a model encoded in XMI 1.2 rather than in XMI 2.1, the user must select a different option when executing the Flock migration strategy. Otherwise, no other changes are required.

## 6.4.3 Comparison with other solutions

At the workshop, solutions to the migration case described in Section 6.4.1 were presented. Each solution was allocated two opponents who highlighted weaknesses of each approach. Following the solution presentations and opposition statements, each solution was scored using the criteria described above: correctness, clarity, conciseness, appropriateness, tool maturity, reproducibility and number of extensions solved. Flock scored the highest average marks for four of seven criteria, and was awarded overall first prize. The remainder of this section discusses the scores in more detail, and summarises the opposition statements for Flock.

### Opposition Statements

The opposition statements highlighted two weaknesses of Flock. Firstly, there is some duplicated code in Listing 6.13: the `migrate Pseudostate to ...` statement

---

[16] `http://www.planet-research20.org/ttc2010/index.php?option=com_community&view=groups&task=viewdiscussion&groupid=4&topicid=20&Itemid=150` (registration required)

| **Response #** | **1** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **Mean** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Correctness | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | |
| Conciseness | 1 | 2 | 2 | 1 | 1 | 1 | 0 | 1 | 2 | 1 | |
| Clarity | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | |
| Extensions | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | |
| Appropriateness | 1 | 2 | 2 | 1 | 2 | 1 | 1 | 2 | 2 | 2 | |
| Tool Maturity | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | |
| Reproducibility | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| Total | 6 | 9 | 9 | 6 | 7 | 6 | 7 | 9 | 9 | 8 | 7.6 |

Table 6.5: TTC scores for Epsilon Flock (unweighted).

appears several times. The duplication exists because Flock only allows one-to-one mappings between original and evolved metamodel types. The conservative copy algorithm would need to be extended to allow one-to-many mappings to remove this kind of duplication.

Secondly, the body of Flock rules are specified in an imperative manner. Consequently, reasoning about the correctness of a migration strategy is arguably more difficult than in languages that use a purely declarative syntax. This point is discussed further in Section 6.5, which considers the limitations of the thesis.

### Scoring

Flock was awarded the overall first prize and scored the highest average marks for five of the seven criteria outlined above. The overall ranking process is first described, and the remainder of the section discusses the score awarded to Flock for each of the criteria.

During the workshop, each tool developer presented their solution. The workshop participants and organisers awarded each solution an individual score for each of the seven criteria outlined above, and a total score (by summing the seven criteria scores). The overall ranking for each solution was calculated by taking the mean of the total scores. For example, Flock was awarded the scores shown in Table 6.5. (Note that Participants #2 and #3 did not award scores to Flock due to a conflict of interest). Appendix D presents the complete set of results.

Although Flock was awarded the overall first prize, few conclusions can be drawn from the rankings. The scores for each criterion were awarded on different scales (e.g. -2 to 2 for conciseness, and 0 to 2 for extensions) and the workshop organisers applied a weight to each criterion before calculating the totals (5 for correctness; 4 for tool maturity; 3 for conciseness, clarity, extensions, and appropriateness; and 2 for reproducibility). Clearly, the relative importance of each criterion may vary between migration cases, and between organisations. Therefore, the remainder of the discussion focusses on the per-criteria scores awarded to Flock and the other tools.

**Correctness** Each tool developer demonstrated the extent to which their solution performed a correct migration of activity diagrams according to the migration semantics described in the case description (Section 6.4.1). The following scores could be awarded: -1 (probably doesn't work at all), 0 (cannot judge), 1 (works for one model), and 2 (works for more than one model). Flock received a mean score of 0.7, and was ranked seventh out of the nine solutions. Migration with Flock is specified with both imperative and declarative language constructs, while many of the other solutions use only declarative language constructs to specify the migration of UML activity diagrams and, hence, more could be said about the correctness of the solutions written in those languages.

**Conciseness** Solutions were awarded one of the following scores for the conciseness of their migration strategies: -2 (very verbose), -1 (quite verbose), 0 (cannot judges), 1 (quite concise), and 2 (very concise). Flock received a mean score of 1.2, and was ranked first out of the nine solutions. Three of the solutions used general purpose languages (such as Java and Prolog), and these were ranked sixth, seventh and ninth. The other solutions used graph or model transformation languages, and, in general, scored more highly than those written in general-purpose languages.

**Clarity** The extent to which the intention of the migration could be determined from the migration strategy was scored on the following scale: -1 (no idea how it works), 0 (some idea how it works), 1 (fully understand how it works). Flock received a mean score of 0.9, was ranked first out of the nine solutions, and there was little variation in the scores awarded to Flock (a score of 1 from eleven of the twelve responses, and a score of 0 from the remaining respondent). Tools tailored to model migration, such as Flock and COPE [Herrmannsdoerfer *et al.* 2009a], and graph transformation languages, such as GrGen [Geiß & Kroll 2007] and MOLA [Kalnins *et al.* 2005], were ranked the highest in this category.

**Appropriateness** The suitability of the tool for migrating activity diagrams was assessed on the following scale: -2 (totally inappropriate), -1 (inappropriate), 0 (neutral), 1 (somewhat appropriate), 2 (perfect fit). Flock received a mean score of 1.6, and was ranked first out of the nine solutions. Again, tools tailored to model migration, such as Flock and COPE [Herrmannsdoerfer *et al.* 2009a], and graph transformation languages, such as GrGen [Geiß & Kroll 2007] and MOLA [Kalnins *et al.* 2005], were ranked the highest in this category.

**Tool maturity** The maturity of each tool was discussed during the solution presentations, and the workshop participants were able to use eight of the nine solutions via a virtual machine. Scores were awarded on the following scale: -1 (prototype), 0 (average), 1 (good). Flock received a mean score of 0.4, and was ranked third out of the nine solutions. Fujaba [Nickel *et al.* 2000] and GrGen [Geiß & Kroll 2007] were ranked first and second in this category, and are established transformation tools that were first reported in the literature in 2000 and 2007 respectively.

**Reproducibility** Each developer was invited to configure a virtual machine with their tool and solution, and the workshop participants were invited to use each of the tools. A score of 1 was awarded if a working virtual machine image was provided by the tool developer, and 0 otherwise. Flock had a mean score of 1, and ranked joint first along with seven of the other tools. The virtual machine image for one of the tools did not work, and it was awarded a mean score of 0.

**Extensions** Three extensions to the core task were described in Section 6.4.1, and a point was awarded for approaching each additional task. Flock was awarded a mean score of 5.4, and ranked first of the nine solutions. Determining the extensions approached by a solution seems to be an objective task, but some tools were awarded different scores for the extensions criterion which is difficult to explain. For instance, the Flock solution (Section 6.4.2) approached two of the three extensions, but some of the participants awarded Flock only 1 point. Rather than analysing the scores, it is perhaps more interesting to note that the Flock solution was the only solution to approach the XMI extension, and similarly for Fujaba [Nickel *et al.* 2000] and the concrete syntax extension. This might indicate that contemporary migration tools can be used to manage realistic metamodel changes, but lack some features that would be desirable in an industrial setting (namely, interoperability with several modelling technologies and co-migration of abstract and concrete syntax).

### 6.4.4 Summary

This section has discussed the way in which Flock was evaluated by participating in the 2010 edition of the Transformation Tools Contest (TTC). Flock was assessed by application to an example of migration from the UML and comparison with eight other model and graph transformation tools. Flock was awarded the overall first prize and ranked first in five of seven categories by the workshop participants and organisers.

In addition to evaluating Flock, the work described in this section provides three further contributions. Firstly, the migration case submitted to TTC 2010, described in Section 6.4.1 provides a real-world example of co-evolution for use in future comparisons of model migration tools. The case is based on the evolution of UML, between versions 1.4 and 2.2. The migration strategy was devised by analysis of the UML specification, and by discussion between workshop participants.

Secondly, the Flock solution to the migration case (Section 6.4.2) demonstrates the way in which a migration strategy can be constructed using Flock. In particular, Section 6.4.2 describes an iterative and incremental development process and indicates that an empty Flock migration strategy can provide a useful starting point for development.

Finally, Section 5.4 claims that Flock supports several modelling technologies. The solution described in Section 6.4.2 demonstrates the way in which Flock can be used to migrate models over two modelling technologies: MDR (XMI 1.x) and EMF (XMI 2.x), and hence supports the claim made in Section 5.4.

## 6.5 Limitations of the Proposed Structures and Processes

The limitations of the research presented in the thesis are now discussed. Some of the shortcomings identified here are elaborated on in Section 7.2, which highlights areas of future work.

**Generality** The thesis research focuses on model-metamodel co-evolution, but, as discussed in Chapter 4, metamodel changes can affect artefacts other than models. Model management operations and model editors are specified using metamodel concepts and, consequently, are affected when a metamodel changes. The work presented in Chapter 5 focuses on migrating models in response to metamodel changes, and does not consider integration with tools for migrating model management operations and model editors. To reduce the effort required to manage the effects of metamodel changes, it seems reasonable to envisage a unified approach that migrates models, model management operations, model editors, and other affected artefacts.

**Reproducibility** The analysis and evaluation presented in Chapters 4 and 6 respectively involved using migration tools to understand and assess their functionality. With the exceptions noted below, the work presented in these chapters is difficult to reproduce and therefore the results drawn are somewhat subjective. On the other hand, multiple approaches to analysis and evaluation have been taken, and the work has been published and subjected to peer review.

Not all of the work in Chapter 4 and 6 is difficult to reproduce. In particular, the evaluation methods used in Chapter 6 are described in detail and a complete set of results are provided in Appendices C and D. In general, the lack of real-world examples of co-evolution restricts the extent to which any work in this area can be considered reproducible.

**Formal semantics** No formal semantics for the conservative copy algorithm (Section 5.4) have been provided. Instead, a reference implementation, Epsilon Flock, was developed, which facilitated comparison with other migration and transformation tools. Without a reference implementation, the evaluation described in Sections 6.2, 6.3 and 6.4 would have been impossible. For Epsilon as a whole, a similar case has been made in favour of a reference implementation over a formal semantics [Kolovos 2009]. For domains where completeness and correctness are a primary concern, a formal semantics would be required before Flock could be applied to manage model-metamodel co-evolution.

## 6.6 Chapter Summary

The work presented in this chapter assessed the structures and processes proposed in Chapter 5. The evaluation has investigated the extent to which the prototypical implementations of the structures and processes increase developer productivity for co-evolution. Factors that affect the efficacy of the proposed structures and processes were identified via the evaluation, and are summarised below. A range of techniques were used for evaluation, and the work presented in this chapter provides a foundation for future work on – and further evaluation of – structures and processes for managing and identifying model-metamodel co-evolution.

Section 6.1 explored and compared the ways in which user-driven co-evolution is performed with and without two of the structures proposed in Chapter 5. The comparison suggested that the way in which models are represented and the way in which conformance is checked affects developer productivity. A model representation that is optimised for human rather than machine use might reduce the likelihood of mistakes, particularly if the developers performing co-evolution lack expertise in manipulating XMI (the canonical model storage representation used by contemporary modelling frameworks). Checking conformance as the model is migrated might prevent errors, particularly if several types of breaking metamodel change are to be resolved, but might be a waste of computational resources when few breaking metamodel changes are to be resolved.

Section 6.2 investigated the conciseness of model migration strategies written using new-target, existing-target and conservative copy source-target relationships. The results indicate that conservative copy yields the most concise model migration strategies, which might lead to increased developer productivity when specifying migration manually. However, other factors that were not assessed in this chapter will affect productivity, such as the learnability of the three source-target relationships. When specifying migration with an operator-based or an inference approach, migration strategies are typically presented to the user to facilitate selection between feasible alternatives. Further evaluation is needed to assess whether conciseness of migration strategies is a desirable quality for operator-based and inference approaches.

Sections 6.3 and 6.4 assessed Epsilon Flock (Section 5.4), a model-to-model transformation language tailored for model migration, by comparison with developer-driven co-evolution tools and model-to-model transformation languages. The comparisons illuminated various considerations for co-evolution. For example, the requirements and maturity of the MDE process can preclude the use of some co-evolution approaches. When an established MDE process is used to develop a system, it might be impossible to use co-evolution approaches that mandate specific tools or techniques, such as operator-based approaches, which require that evolution is expressed with a specialised metamodel editor.

In addition to the evaluation described in this chapter, the work presented in this thesis has been subject to peer review by the academic and Eclipse communities. The thesis research has been published in 3 workshop papers, 2 European conference papers and 4 international conferences papers. Epsilon HUTN and Epsilon Flock (Chapter 5) were contributed to the Epsilon project, a component of the research incubator for the Eclipse Modeling Project (EMP), which is arguably the most active MDE community at present. EMP's research incubator hosts a limited number of participants, selected through a rigorous process and contributions made to the incubator undergo regular technical review.

Evaluation of the research hypothesis has highlighted areas in which the structures and processes proposed in Chapter 5 should be improved, and has motivated plans for future work, which are described in Section 7.2. Additionally, the work presented in this chapter has identified areas in which further evaluation might be carried out. In particular, comprehensive user studies are needed to assess the usability and learnability of co-evolution tools, and to further evidence the benefits of co-evolution tools in terms of increased developer productivity.

# Chapter 7

# Conclusions

This thesis has investigated software evolution – a key and costly development activity in software engineering [Moad 1990] – in the context of Model-Driven Engineering (MDE), a state-of-the art approach to software engineering. While MDE promises increased developer productivity [Watson 2008] and increased portability of software systems [Frankel 2002], it also poses several challenges that threaten its adoption. For example, identifying and managing evolutionary change in the context of MDE presents many open research challenges [Mens & Demeyer 2007]. The thesis research has contributed to these research challenges and has explored the following research hypothesis:

> *In existing MDE projects, the evolution of MDE development artefacts is typically managed in an ad-hoc manner with little regard for re-use. Dedicated structures and processes for managing evolutionary change can be designed by analysing evolution in existing MDE projects. Furthermore, supporting those dedicated structures and processes in contemporary MDE environments is beneficial in terms of increased productivity for software development activities pertaining to the management of evolutionary change.*

To explore the thesis hypothesis, the following research objectives were defined.

1. Identify and analyse the evolution of MDE development artefacts in existing projects.
2. Investigate the extent to which existing structures and processes can be used to manage the evolution of MDE development artefacts.
3. Propose and develop new structures and processes for managing the evolution of MDE development artefacts, and integrate those structures and processes with a contemporary MDE development environment.
4. Evaluate the proposed structures and processes for managing evolutionary change, particularly with respect to productivity.

The remainder of this chapter summarises the contributions of the thesis in relation to the thesis hypothesis and research objectives, and gives a brief description of and motivation for several potential extensions to the thesis research.

## 7.1 Contributions of the Thesis Research

The primary contributions of the thesis are summarised below, and the remainder of this section discusses each contribution in turn.

- Chapter 4 presented an analysis of evolution in existing MDE projects, which indicated ways in which models, metamodels and model management operations evolve, highlighted model-metamodel co-evolution as a commonly-occuring software evolution activity in MDE projects and led to a categorisation of existing approaches to managing model-metamodel co-evolution.
- Chapter 5 described the design and implementation of structures and processes for performing model-metamodel co-evolution, which included a metamodel-independent syntax for managing non-conformant models, a textual modelling notation for manually migrating models, and a model-to-model transformation language tailored for migration. The proposed structures and processes are interoperable with EMF (Section 2.3.1), arguably the most widely-used contemporary MDE modelling framework.
- Chapter 6 detailed the evaluation of the proposed structures and processes using quantitive measurements, expert evaluation and application to large, independent examples of co-evolution and explored the extent to which the proposed structures and processes are beneficial in terms of increased developer productivity.

### 7.1.1 Investigation of Evolution in MDE Projects

The way in which evolution occurs and is managed in existing MDE projects was analysed in Chapter 4. The analysis investigated two types of evolutionary change, model-metamodel co-evolution and model synchronisation, which were identified from the review presented in Chapter 3. For the MDE projects considered in Chapter 4, several suitable model-metamodel co-evolution examples – and no suitable model synchronisation examples – were located, and consequently the remainder of the thesis focused on model-metamodel co-evolution.

The co-evolution examples were used to identify the differences between existing approaches to managing model-metamodel co-evolution, and to investigate the way in which model-metamodel co-evolution is managed in existing MDE projects. The investigation led to the definition of two distinct approaches to managing model-metamodel co-evolution in MDE projects, *user-driven* and *developer-driven* and to a categorisation of existing approaches, which was published in [Rose *et al.* 2009b] and has since been used and extended in several papers, including [Jurack & Mantz 2010, Méndez *et al.* 2010].

### 7.1.2 Structures and Processes for Managing Co-evolution

The analysis of existing MDE projects presented in Chapter 4 highlighted challenges for identifying and managing co-evolution. Managing co-evolution in a user-driven manner, for instance, is particularly challenging in contemporary MDE modelling environments because conformance is enforced implicitly and models and metamodels are kept separate. Similarly, the variation in programming and transformation languages typically used to specify model migration presents a challenge for comparing existing approaches to developer-driven co-evolution approaches. Moreover, none of the languages typically used have been tailored for the specific requirements of model migration. This thesis contributes structures and processes that seek to address the challenges summarised above.

#### Metamodel-Indepenent Syntax

Contemporary MDE modelling frameworks cannot be used to load non-conformant models. Consequently, model migration cannot be performed using the structures typically available in contemporary MDE modelling environments, such as model editors and model management operations. The metamodel-independent syntax, introduced in Chapter 5, is a proposed extension to contemporary MDE modelling frameworks that facilitates the loading of non-conformant models and provides services for reporting conformance problems.

The metamodel-independent syntax underpins the implementation of two further structures, the textual modelling notation described below, and the automatic conformance checking service of Concordance [Rose *et al.* 2010c].

#### Textual Modelling Notation

When a small number of models are to be migrated, the effort required to specify an executable migration strategy might not be justifiable. Instead, models can be migrated by editing models by hand. The textual modelling notation, presented in Chapter 5, provides a notation for editing models in contemporary MDE development environments. The notation proposed in this thesis adopts the Human-Usable Textual Notation (HUTN) [OMG 2004], a standard notation for textual modelling proposed by the Object Management Group (OMG)[1]. The implementation of HUTN introduced in Section 5.2, Epsilon HUTN, is the sole reference implementation of the HUTN standard, and was published in [Rose *et al.* 2008a].

During user-driven co-evolution, model editors cannot be used for migration and editing models in their underlying storage representation, which will not have been optimised for human use, can be error-prone and time consuming. The textual modelling notation introduced in Chapter 5, Epsilon HUTN, provides an alternative to editing models in their underlying storage representation.

---

[1] `http://www.omg.org`

### A Process for User-Driven Co-evolution

The analysis of existing MDE projects highlighted several projects in which model migration was performed using user-driven co-evolution techniques, and yet no existing work sought to address the specific requirements of user-driven co-evolution. Contemporary MDE modelling environments typically enforce conformance in an implicit manner, and cannot be used to load non-conformant models. Consequently, user-driven co-evolution is an iterative, error-prone and time-consuming task, because model editors and model management operations cannot be used to assist migration.

A typical process for performing user-driven co-evolution involves performing model migration by repeatedly switching between a model editor (which reports conformance problems) a text editor (in which conformance is reconciled by the user). Chapter 6 proposes an alternative process in which conformance reporting and reconciliation occur in the same environment, using the metamodel-independent syntax and textual modelling notation described above. The alternative process was published in [Rose *et al.* 2009a].

### Epsilon Flock: A Model Migration Language

In addition to the structures and processes for performing user-driven co-evolution, the thesis also contributes a structure dedicated to developer-driven co-evolution, a model migration language termed *Epsilon Flock*. The analysis performed in Chapter 4 showed that model migration is often specified with a model-to-model transformation language or with a general purpose programming language, and that these languages are not tailored to the specific requirements of model migration. The investigation presented in Chapter 5, highlighted that a language tailored for model migration might provide several benefits over repurposing an existing language to specify model migration. Flock was designed and implemented to explore the extent to which a language tailored for model migration might increase developer productivity. The investigation of languages for model migration and the design and implementation of Flock were published in [Rose *et al.* 2010f].

Flock contributes a novel mechanism for relating source and target model elements termed *conservative copy*, which is a hybrid of the two existing mechanisms used to relate source and target model elements in contemporary model-to-model transformation languages. Conservative copy automatically copies to the target model every source model element that conforms to the target metamodel, and does not copy to the target model source model elements that do not conform to the target metamodel.

## 7.1.3 Evaluation of Structures and Processes

The structures and processes introduced in this thesis have been evaluated using a variety of techniques, including a quantitive comparison, an expert evaluation and comparison to related processes and structures. Existing work on evolutionary change in MDE has typically been evaluated with a case study (such as in [Sprinkle 2003]) or by demonstration (such as in [Cicchetti 2008]), and few papers in the area report the strengths and weaknesses of proposed approaches, or seek to contextualise their contentions. The evaluation presented in Chapter 6 is a contribution in its own right as it presents several alternative evaluation techniques, including the first expert evaluation of model migration tools, and seeks to identify situations in which the proposed structures and processes are both effective and ineffective.

## 7.2 Future Work

In the context of a doctoral thesis, it is impossible to thoroughly investigate many of the issues raised in exploring the thesis hypothesis. Below, several potential extensions to the research presented in this thesis are identified, and any initial work in those areas is described.

### 7.2.1 Further Evaluation

The extent to which the structures and processes introduced in Chapter 5 increase developer productivity for co-evolution has been explored via expert evaluation and comparison to related work. Assessing the way in which the proposed structures and processes affect productivity is challenging due to the number of factors that affect productivity. In practice, for example, the proposed structures and processes would be used by developers with different expertise, and together with other tools and techniques. Evaluating the way in which software evolution is identified and managed in practice using comprehensive case and user studies would be likely to allow stronger claims to be made about the efficacy of the proposed structures and processes. Given the time constraints of a doctoral thesis, comprehensive case and user studies were not feasible, and hence are desirable extensions to the thesis research. A key first step would be to establish a common vocabulary for discussing software evolution activities in the context of MDE, which would facilitate the comparison of user experiences.

### 7.2.2 Extensions to the Model Migration Language

The model migration language proposed and implemented in Chapter 5, Epsilon Flock, makes idiomatic commonly occurring patterns of model migration. The evaluation presented in Chapter 6 suggested that migration strategies are often more concise when specified with Flock rather than when specified with contemporary model-to-model transformation languages. The evaluation also highlighted a limitation in the implementation of Flock and demonstrated further patterns that might be captured by model migration languages.

Addressing these issues would further improve the conciseness and re-usability of model migration strategies written in Flock and, hence, is an obvious area of future work. In particular, one language construct controls two concerns in the current implementation of Flock, and introducing separate language constructs for each concern could increase the potential for re-use between model migration rules. Applying Flock to the co-evolution examples used for evaluation highlighted further model migration patterns that might be made idiomatic in the language. For example, in situations where conservative copy can have side-effects, it may be desirable to afford more control of the copying algorithm via, for example, an `ignore` keyword that identifies values that should not be automatically copied.

### 7.2.3 Unifying Co-evolution Approaches

The thesis research has focused on one type of software evolution, model-metamodel co-evolution. Many further types of evolution occur in practice, including model refactoring and model synchronisation, which were discussed in Chapter 3. Changes to a metamodel affect not only models but also model management operations, such as model transformations. When changes are propagated from a metamodel to a model during migration, further artefacts might be impacted as an indirect consequence of the metamodel evolution.

The use of distinct structures and processes for each type of evolution poses usability challenges relating to the interoperability of tools and increased training effort. Seeking, instead, a unified approach to managing evolution might address these challenges, and presents an interesting opportunity for future work. Integrating the thesis research with approaches for managing other types of co-evolution, such as transformation-metamodel co-evolution, is one way in which the formulation of a unified approach might proceed. To this end, an outline for integrating model-metamodel and transformation-metamodel co-evolution approaches has proposed in collaboration with Anne Etien, an Associate Professor at the Université Lille, and published in [Rose *et al.* 2010a].

### 7.2.4 Higher-Order Migration

In model transformation, a higher-order transformation consumes or produces a model transformation. Higher-order transformation has been used to generate model migration strategies [Cicchetti 2008, Garcés *et al.* 2009], and to compose and analyse transformations [Tisi *et al.* 2009]. Similarly, higher-order migration might be used effectively for migrating model transformation specifications between similar model transformation languages, and for migrating model management operations in response to changes to their specification language. For example, higher-order migration might be applied to migrate model migration strategies between different types of transformation language, such as from a new-target to a conservative copy language.

### 7.2.5 Genericity

Chapter 6 identified a lack of metamodel-independent re-use as one of the primary weaknesses of Flock compared to related approaches. In Flock, model migration strategies are specified in terms of metamodel concepts, and consequently, the extent to which code can be re-used across migration strategies is reduced. By mixing model management languages with ideas from generic programming, one way in which model management operations can be specified in a manner that is independent of their metamodel has been identified [de Lara & Guerra 2010]. Applying these ideas to Flock would facilitate increased re-use across model migration strategies, and address a primary weakness of Flock.

## 7.3 Closing Remarks

Building the systems demanded by society now and in the future will require new approaches to software engineering [Selic 2003]. MDE is a state-of-the-art, principled approach to software engineering, and promises many benefits particularly with respect to the portability and maintainability of software systems [Kleppe *et al.* 2003, Frankel 2002]. While MDE shows promise, its success is reliant on the availability of mature and powerful tools. Such tools are beginning to emerge, but typically fail to address concerns that affect their applicability to the engineering of large and complex software systems, such as scalability and the cost of systems evolution.

The work presented in this thesis has demonstrated a systematic method for identifying challenges for software evolution in typical MDE processes, proposed structures and processes for addressing those challenges, and evaluated the structures and processes by comparison to related work and by application to real-world examples of evolution.

## Appendix A

# Code Listings

This appendix provides complete versions of the code listings discussed in this thesis. Specifically, this appendix presents listings for migrating Petri nets with Ecore2Ecore (Section 5.3.2), and presents the model management operations used to implement Epsilon HUTN (Section 5.2).

## A.1 Migrating Petri Nets with Ecore2Ecore

The code in Listings A.1 and A.2 demonstrates the way in which the Ecore2Ecore tool [Hussey & Paternostro 2006] can be used to perform model migration, using the Petri nets example described in Section 5.3.1.

```
package lit_petriNets.resources;

import java.io.InputStream;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Map;
import java.util.Map.Entry;

import lit_petriNets.Lit_petriNetsFactory;
import lit_petriNets.Lit_petriNetsPackage;
import lit_petriNets.PTArc;
import lit_petriNets.Place;
import lit_petriNets.TPArc;
import lit_petriNets.Transition;

import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.eclipse.emf.ecore.resource.Resource;
```

```
import org.eclipse.emf.ecore.util.FeatureMap;
import org.eclipse.emf.ecore.xmi.XMLResource;
import org.eclipse.emf.ecore.xmi.impl.BasicResourceHandler;
import org.eclipse.emf.ecore.xml.type.AnyType;

public class PetriNetsResourceHandler extends BasicResourceHandler {

 @Override
 public void postLoad(XMLResource resource, InputStream inputStream,
     Map<?, ?> options) {
  final Map<EObject, AnyType> extMap = resource.
      getEObjectToExtensionMap();

  for(Entry<EObject, AnyType> entry : extMap.entrySet()) {
   handleUnknownData(entry.getKey(), entry.getValue());
  }
 }

 private void handleUnknownData(EObject eObj, AnyType unknownData) {
  handleUnknownFeatures(eObj, unknownData.getMixed());
  handleUnknownFeatures(eObj, unknownData.getAnyAttribute());
 }

 private void handleUnknownFeatures(EObject owner, FeatureMap
     featureMap) {
  for (Iterator<FeatureMap.Entry> iter = featureMap.iterator(); iter.
      hasNext();) {
   final FeatureMap.Entry entry = iter.next();

   if (isTransition(owner)) {
    if (isCollectionOfPlaces(entry.getValue(), owner.eResource())) {
     final Transition transition = (Transition)owner;
     final Collection<Place> places = toCollectionOfPlaces(entry.
         getValue(), owner.eResource());

     if (isSrc(entry.getEStructuralFeature())) {
      migrateSrc(transition, places);

     } else if (isDst(entry.getEStructuralFeature())) {
      migrateDest(transition, places);
     }

```

```
          iter.remove();

        } else {
          System.err.println("Not␣a␣collection␣of␣places:␣" + entry.getValue());
        }
      } else {
        System.err.println("Not␣a␣transition:␣" + owner);
      }
    }
  }

  private boolean isTransition(EObject eObject) {
    return Lit_petriNetsPackage.eINSTANCE.getTransition().isInstance(eObject);
  }

  private boolean isSrc(EStructuralFeature feature) {
    return "src".equals(feature.getName());
  }

  private boolean isDst(EStructuralFeature feature) {
    return "dst".equals(feature.getName());
  }

  private boolean isCollectionOfPlaces(Object value, Resource resource) {
    final String[] uriFragments = ((String)value).split("␣");

    for (String uriFragment : uriFragments) {
      final EObject eObject = resource.getEObject(uriFragment);

      if (eObject == null || !Lit_petriNetsPackage.eINSTANCE.getPlace().isInstance(eObject))
        return false;
    }

    return true;
  }

  private Collection<Place> toCollectionOfPlaces(Object value, Resource resource) {
```

```
      final String[] uriFragments = ((String)value).split("␣");

      final Collection<Place> places = new LinkedList<Place>();

      for (String uriFragment : uriFragments) {
        places.add((Place)resource.getEObject(uriFragment));
      }

      return places;
    }

    private void migrateSrc(Transition owner, Collection<Place> sources) {
      for (Place source : sources) {
        final PTArc arc = Lit_petriNetsFactory.eINSTANCE.createPTArc();
        arc.setSrc(source);
        arc.setDst(owner);
        arc.setNet(owner.getNet());
      }
    }

    private void migrateDest(Transition owner, Collection<Place> destinations) {
      for (Place destination : destinations) {
        final TPArc arc = Lit_petriNetsFactory.eINSTANCE.createTPArc();
        arc.setSrc(owner);
        arc.setDst(destination);
        arc.setNet(owner.getNet());
      }
    }
}
```

Listing A.1: Resource handler for migrating Petri net models.

```
package lit_petriNets.resources;

import java.util.Map;

import lit_petriNets.Lit_petriNetsPackage;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
```

```java
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.eclipse.emf.ecore.util.ExtendedMetaData;
import org.eclipse.emf.ecore.xmi.XMIResource;
import org.eclipse.emf.ecore.xmi.XMLResource;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
import org.eclipse.emf.mapping.ecore2xml.Ecore2XMLPackage;
import org.eclipse.emf.mapping.ecore2xml.Ecore2XMLRegistry;
import org.eclipse.emf.mapping.ecore2xml.impl.Ecore2XMLRegistryImpl;
import org.eclipse.emf.mapping.ecore2xml.util.Ecore2XMLExtendedMetaData
    ;

public class PetriNetsFactoryImpl extends XMIResourceFactoryImpl {

  public static final String BEFORE_NS_URI = "lit_petriNets";
  public static final String AFTER_PLATFORM_URI = "platform:/plugin/
      petrinets/model/After.ecore";
  public static final String BEFORE_PLATFORM_URI = "platform:/plugin/
      petrinets/model/Before_2_After.ecore2xml";

  private ExtendedMetaData extendedMetaData;


  public Resource createResource(URI uri) {
    final XMIResource resource = (XMIResource) super.createResource(uri)
        ;

    final Map<Object, Object> defaultLoadOptions = resource.
        getDefaultLoadOptions();
    defaultLoadOptions.put(XMLResource.OPTION_EXTENDED_META_DATA,
        getExtendedMetaData());
    defaultLoadOptions.put(XMLResource.OPTION_RECORD_UNKNOWN_FEATURE,
        Boolean.TRUE);
    defaultLoadOptions.put(XMLResource.OPTION_RESOURCE_HANDLER, new
        PetriNetsResourceHandler());

    return resource;
  }

  private ExtendedMetaData getExtendedMetaData() {
    if(extendedMetaData == null) {
      final ResourceSet resourceSet = new ResourceSetImpl();
```

```
      final EPackage.Registry ePackageRegistry = resourceSet.
          getPackageRegistry();

      ePackageRegistry.put(BEFORE_NS_URI, Lit_petriNetsPackage.eINSTANCE)
          ;
      ePackageRegistry.put(AFTER_PLATFORM_URI, Lit_petriNetsPackage.
          eINSTANCE);

      Ecore2XMLRegistry ecore2xmlRegistry = new Ecore2XMLRegistryImpl(
          Ecore2XMLRegistry.INSTANCE);
      ecore2xmlRegistry.put(BEFORE_NS_URI,
        EcoreUtil.getObjectByType(
          resourceSet.getResource(URI.createURI(BEFORE_PLATFORM_URI),
            true).getContents(),
            Ecore2XMLPackage.Literals.XML_MAP));

      extendedMetaData = new Ecore2XMLExtendedMetaData(ePackageRegistry,
          ecore2xmlRegistry);
    }
    return extendedMetaData;
  }
}
```

Listing A.2: Resource factory for migrating Petri net models.

## A.2 Model Management Operations for Epsilon HUTN

The code listings in this section demonstrate the way in which Epsilon HUTN (Section 5.2) is implemented, and are taken from Epsilon HUTN v0.8.0 (which was released in November 2010).

### A.2.1 AST Model to Intermediate Model

The ETL transformation in Listing A.3 transforms a model that conforms to the AST metamodel (Figure 5.15) into a model that conforms to the metamodel-independent syntax (Section 5.1).

```
pre {
  var EmfTool := new Native('org.eclipse.epsilon.emc.emf.tools.EmfTool');

  var ast  := AntlrAst!Ast.allInstances().first();
  var config := Config!Configuration.allInstances().first();

  var spec := Intermediate!Spec.allInstances().first();
```

```

 for (root in ast.roots) {
  if (AntlrAst!NameNode.isType(root) and root.text.toLowerCase() = ’@spec’) {
   for (child in root.children) {

    -- process @Spec.model package
    if (child.text.toLowerCase() = ’model’) {
     var modelFileAttribute := child.children.selectOne(n : AntlrAst!NameNode | n.text = ’file’);

     if (modelFileAttribute.isDefined()) {
      var modelFileValue := modelFileAttribute.children.selectOne(t : AntlrAst!TextualValueNode | true);

      if (modelFileValue.isDefined()) {
       spec.modelFile = modelFileValue.getValue();
      }
     }

    -- process all other @Spec packages as metamodel specifications
    } else {

     for (grandchild in child.children.select(n : AntlrAst!NameNode | n.text.toLowerCase() = ’nsuri’)) {
      for (valueNode in grandchild.children.select(t : AntlrAst!TextualValueNode | true)) {
       var nsUri := new Intermediate!NsUri;
       nsUri.value := valueNode.getValue();
       nsUri.addTraceabilityInfo(valueNode.parent);
       spec.nsUris.add(nsUri);
      }
     }
    }
   }
  }
 }
}

rule NameNode2PackageObject
 transform n : AntlrAst!NameNode
 to p : Intermediate!PackageObject {

 guard : n.parent.isUndefined() and n.text.toLowerCase() <> ’@spec’

 spec.objects.add(p);

 p.type := n.text;
```

```
 p.addIdentifier(n, false);
 p.addTraceabilityInfo(n);

 if (not spec.nsUris.isEmpty()) {
  for (nsUri in spec.nsUris) {
   var package := EmfTool.getEPackage(nsUri.value);

   if (package.isDefined()) {
    p.metamodel.add(package);
   }
  }
 }

 for (child in n.children.select(n : AntlrAst!NameNode | true)) {
  p.classObjects.add(child.equivalent());
 }

 -- Transform association blocks and infix associations
 n.children.forAll(a : AntlrAst!AssociationInstanceNode | a.createReferences(p))
    ;
}

rule NameNode2ClassObject
 transform n : AntlrAst!NameNode
 to c : Intermediate!ClassObject {

 guard : n.isClass() and n.parent.text.toLowerCase() <> '@spec'

 createClassObject(c, n);
}

operation AntlrAst!AssociationInstanceNode createReferences(package :
    Intermediate!PackageObject) {

 var slot : Intermediate!ReferenceSlot;
 var currentChildIsSource := true;

 for (ref in self.children) {
  var cls := Intermediate!ClassObject.allInstances().selectOne(c|c.identifier =
       ref.children.first().text.stripQuotes());

  if (cls.isUndefined()) {
   cls := createClassObject(ref);
   package.classObjects.add(cls);
  }

  if (currentChildIsSource) {
```

```
      slot := cls.slots.selectOne(r : Intermediate!ReferenceSlot | r.feature =
          self.text);

      if (slot.isUndefined()) {
        slot := new Intermediate!ReferenceSlot;
        slot.feature := self.text;
        cls.slots.add(slot);
      }

    } else {
      slot.values.add(cls.identifier);
    }

    currentChildIsSource := not currentChildIsSource;
  }
}



operation AntlrAst!Node isClass() : Boolean {
  if (self.parent.isUndefined()) {
    return false;
  } else {
    return ast.roots.includes(self.parent);
  }
}

operation Intermediate!ModelElement addTraceabilityInfo(node : AntlrAst!Node) :
     Intermediate!ModelElement {
  self.line := node.line;
  self.col := node.column;
  return self;
}

operation Intermediate!Object addIdentifier(node : AntlrAst!Node) {
  self.addIdentifier(node, true);
}

operation Intermediate!Object addIdentifier(node : AntlrAst!Node, infer :
     Boolean) {
  var identifierNode : AntlrAst!Node := node.children.select(t : AntlrAst!
      TextualValueNode | true).first();

  if (identifierNode.isDefined()) {
    self.identifier := identifierNode.text.stripQuotes();

  } else if (infer) {
```

```
    var nameSlot : Intermediate!AttributeSlot := self.slots.select(s: Intermediate!AttributeSlot | s.feature = self.getIdentifierAttributeName()).first();

    if (nameSlot.isDefined() and nameSlot.values.size() = 1) {
      self.identifier := nameSlot.values.first();
    }
  }
}

operation Intermediate!ClassObject inferAttributeValueFromIdentifier() {
  if (self.identifier.isDefined() and self.getIdentifierAttributeName().isDefined() and not self.slots.exists(s : Intermediate!AttributeSlot | s.feature = self.getIdentifierAttributeName())) {
    var slot := new Intermediate!AttributeSlot;
    slot.feature := self.getIdentifierAttributeName();
    slot.values.add(self.identifier);
    self.slots.add(slot);
  }
}

operation Intermediate!ClassObject getIdentifierAttributeName() : String {
  var idRule : Config!IdentifierRule := config.rules.select(r : Config!IdentifierRule | r.classifier = self.type).first();

  if (config.isDefined() and idRule.isDefined()) {
    return idRule.attribute;
  }
}

operation Intermediate!ClassObject addDefaultValuesForAttributes() {
  var classifierLevelAttributes := AntlrAst!ClassifierLevelAttributeNode.allInstances().select(cla | cla.children.first().text = self.type).collect(cla | cla.children.first());

  for (cla in classifierLevelAttributes) {
    var attribute     := cla.children.first().text;
    var defaultValues := cla.children.first().children.collect(node | node.text.stripQuotes());

    if (not self.slots.exists(s : Intermediate!AttributeSlot | s.feature = attribute)) {
      self.slots.add(createSlotFor(defaultValues, attribute).addTraceabilityInfo(cla));
    }
  }

```

```
  var defaultValueRules := config.rules.select(r : Config!DefaultValueRule | r.
      classifier = self.type);

  for (defaultValueRule in defaultValueRules) {
    if (not self.slots.exists(s : Intermediate!AttributeSlot | s.feature =
        defaultValueRule.attribute)) {
      self.slots.add(createSlotFor(defaultValueRule.defaultValue, defaultValueRule
          .attribute));
    }
  }
}

-- Default Value Rules --
operation createSlotFor(defaultValue : String, attribute : String) :
    Intermediate!Slot {
  var defaultValues := new Sequence;
  defaultValues.add(defaultValue);

  return createSlotFor(defaultValues, attribute);
}

operation createSlotFor(defaultValues : Collection, attribute : String) :
    Intermediate!Slot {
  var slot : Intermediate!Slot := defaultValues.first().inferType().toSlot();

  slot.feature := attribute;

  defaultValues.forAll(value | slot.values.add(value.inferType()));

  return slot;
}

operation String inferType() : Any {
  -- Boolean?
  if (self = 'true') {
    return true;

  } else if (self = 'false') {
    return false;

  -- Int?
  } else if (self.isInteger()) {
    return self.asInteger();

  -- Real?
  } else if (self.isReal()) {
    return self.asFloat();
```

```

  -- String
  } else {
    return self;
  }
}

operation String stripQuotes() : String {
  var result : String := self;

  if (result.startsWith(’"’)) {
    result := result.substring(1);
  }

  if (result.endsWith(’"’)) {
    result := result.substring(0, result.length() - 1);
  }

  return result;
}

operation Intermediate!ClassObject addAdjective(a : AntlrAst!AdjectiveNode) {
  var slot := new Intermediate!AttributeSlot;
  slot.feature := a.getFeature();
  slot.values.add(a.getValue());

  slot.addTraceabilityInfo(a);

  self.slots.add(slot);
}

operation AntlrAst!AdjectiveNode getFeature() : String {
  if (self.text.startsWith(’˜’) or self.text.startsWith(’#’)) {
    return self.text.substring(1);

  } else {
    return self.text;
  }
}

operation AntlrAst!AdjectiveNode getValue() : Boolean {
  return not self.text.startsWith(’˜’);
}

operation Intermediate!ClassObject addAttribute(n : AntlrAst!NameNode) {
  var slot := self.findSlot(n.text);

```

```
  if (slot.isUndefined() or slot.isDifferentTypeTo(n.children.first().toSlot()))
      {
    slot := n.children.first().toSlot();
    slot.feature := n.text;
  }

  if (Intermediate!ReferenceSlot.isType(slot)) {
    for (valueNode in n.children) {
      slot.values.add(valueNode.getValue());
    }

  } else {
    for (valueNode in n.children) {
      if (valueNode.text.isDefined()) {
        slot.values.add(valueNode.getValue());
      }
    }
  }

  slot.addTraceabilityInfo(n);
  self.slots.add(slot);
}

post {
  Intermediate!AttributeSlot.all.forAll(s|s.coerceValues());
}

operation Intermediate!Slot isDifferentTypeTo(other : Intermediate!Slot) {
  return self.eClass.name <> other.eClass.name;
}

-- Nulls --
operation AntlrAst!NullNode toSlot() : Intermediate!Slot {
  return new Intermediate!AttributeSlot;
}

-- Booleans --
operation Boolean toSlot() : Intermediate!Slot {
  return new Intermediate!AttributeSlot;
}

operation AntlrAst!TrueNode toSlot() : Intermediate!Slot {
  return new Intermediate!AttributeSlot;
}

operation AntlrAst!TrueNode getValue() : Boolean {
  return true;
```

```
}

operation AntlrAst!FalseNode toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation AntlrAst!FalseNode getValue() : Boolean {
 return false;
}

-- Strings --
operation String toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation AntlrAst!TextualValueNode toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation AntlrAst!TextualValueNode getValue() : String {
 return self.text.stripQuotes();
}

-- Numbers --
operation Integer toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation Real toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation String isReal() : Boolean {
 return '.'.isSubstringOf(self);
}

operation AntlrAst!NumericValueNode toSlot() : Intermediate!Slot {
 return new Intermediate!AttributeSlot;
}

operation AntlrAst!NumericValueNode getValue() : Any {
 if (self.text.isReal()) {
  return self.text.asDouble();
 } else {
  return self.text.asInteger();
 }
}
```

```

-- References
operation AntlrAst!ReferenceNode toSlot() : Intermediate!Slot {
 return new Intermediate!ReferenceSlot;
}

operation AntlrAst!ReferenceNode getValue() : Any {
 var value = self.children.first().text.stripQuotes();

 if (value.isExternalObjectReference()) {
    var base = Intermediate!Spec.all.first.sourceFile;
   value = EmfTool.resolveURI(value, base);
 }

 return value;
}

operation String isExternalObjectReference() : Boolean {
 return self.contains("#");
}

-- Containment and Enumerations --
operation AntlrAst!NameNode toSlot() : Intermediate!Slot {
 if (self.children.isEmpty()) {
   return new Intermediate!AttributeSlot;

 } else {
   return new Intermediate!ContainmentSlot;
 }
}

operation AntlrAst!NameNode getValue() : Any {
 if (self.children.isEmpty()) {
   return self.text;

 } else {
   return createClassObject(self);
 }
}

operation createClassObject(n : AntlrAst!Node) : Intermediate!ClassObject {
 var c := new Intermediate!ClassObject;
 createClassObject(c, n);
 return c;
}

operation createClassObject(c : Intermediate!ClassObject, n : AntlrAst!Node) {
```

```
  c.type := n.text;
  c.addTraceabilityInfo(n);

  n.children.forAll(a : AntlrAst!AdjectiveNode | c.addAdjective(a));
  n.children.forAll(n : AntlrAst!NameNode | c.addAttribute(n));

  c.addIdentifier(n);
  c.inferAttributeValueFromIdentifier();
  c.addDefaultValuesForAttributes();
}
```

Listing A.3: Transforming AST models to intermediate models with ETL.

### A.2.2 Intermediate Model Validation

The EVL constraints in Listing A.4 are executed on an intermediate model and a target model, and produce a set of consistency problems. Listing A.4 includes the syntactic constraints of HUTN (e.g. all identifiers must be unique) and the conformance constraints defined for the metamodel-independent syntax in Section 5.1.

```
pre {
  var EmfTool := new Native(’org.eclipse.epsilon.emc.emf.tools.EmfTool’);
}

context Object {
  constraint IdentifiersMustBeUnique {
    guard: self.identifier.isDefined()
    check: self.identifier.isUniqueIdentifier()
    message: ’Duplicate identifier: ’ + self.identifier
  }
}

context ClassObject {
  constraint ClassifierMustExist {
    guard: hasSpecificMetamodel()
    check: self.hasEClass() or Metamodel!EClassifier.allInstances().select(c|c.
        name = self.type).size() = 1
    message: ’Unrecognised classifier: ’ + self.type
  }

  constraint ClassifierMustBeClass {
    guard: self.satisfies(’ClassifierMustExist’)
    check: self.hasEClass()
    message: ’Cannot instantiate the enumeration or data type: ’ + self.type
  }

  constraint ClassMustNotBeAbstract {
    guard: self.satisfies(’ClassifierMustBeClass’)
```

```
    check: not self.toClass().isAbstract()
    message: ’Cannot instantiate the abstract class: ’ + self.type
  }

  constraint ClassMustSpecifyRequiredReferences {
    guard: self.satisfies(’ClassMustNotBeAbstract’)
    check: self.getAllReferencesThatRequireAValueButDontHaveOne().isEmpty()
    message: self.identifier + ’ must specify a value for the following reference features: ’ + self.getAllReferencesThatRequireAValueButDontHaveOne().collect(f|f.name).toString()

    fix {
      title : ’Infer empty instances’
      do {
        for (reference in self.getAllReferencesThatRequireAValueButDontHaveOne()) {
          -- An instance can be inferred if no values are required
          if (reference.aValueCanBeInferred()) {
            var instance := new Intermediate!ClassObject;
            instance.type := reference.eType.name;

            var slot := new Intermediate!ContainmentSlot;
            slot.feature := reference.name;
            slot.values.add(instance);

            self.slots.add(slot);
          }
        }
      }
    }
  }
}

context Slot {
  constraint FeatureMustExist {
    guard: hasSpecificMetamodel() and self.owner.isTypeOf(ClassObject) and self.owner.toClass().isDefined() and self.feature.isDefined()
    check: self.owner.toClass().eAllStructuralFeatures.select(c|c.name = self.feature).size() = 1
    message: ’Unrecognised feature: ’ + self.feature
  }

  constraint FeatureMustBeChangeable {
    guard: self.satisfies(’FeatureMustExist’)
    check: self.getEStructuralFeature().changeable
    message: ’Feature ’ + self.getEStructuralFeature().name + ’ is not changeable’
```

```
  }

  constraint MustBeTypeCompatibleWithFeature {
    guard: self.satisfies('FeatureMustExist')
    check: self.typeCompatibleWith(self.getEStructuralFeature())
    message: 'Expected ' + self.getEStructuralFeature().eType.name + ' for: ' +
         self.feature
  }

  constraint SingleValuedFeatureCannotTakeMultipleValues {
    guard: self.satisfies('FeatureMustExist')
    check: self.getEStructuralFeature().isMany() or self.values.size = 1
    message: 'Multiple values not permitted for: ' + self.feature
  }
}

context ReferenceSlot {
  constraint FeatureMustBeReference {
    guard: self.satisfies('MustBeTypeCompatibleWithFeature')
    check: not self.getEStructuralFeature().isContainment
    message: 'A reference value was specified for the containment feature ' +
           self.owner.type + '#' + self.feature + '.'
  }

  constraint ReferencedIdentifiersMustExist {
    guard: self.satisfies('FeatureMustBeReference')
    check: self.values.forAll(i|i.isRecognisedIdentifier())
    message: self.values.selectOne(i|not i.isRecognisedIdentifier()).getMessage()

  }
}

context ContainmentSlot {
  constraint FeatureMustBeContainment {
    guard: self.satisfies('MustBeTypeCompatibleWithFeature')
    check: self.getEStructuralFeature().isContainment
    message: 'A contained object was specified for the non-containment feature '
         +
           self.owner.type + '#' + self.feature + '.'
  }
}

operation hasSpecificMetamodel() : Boolean {
  return Spec.allInstances().at(0).nsUris.notEmpty();
}

operation String isUniqueIdentifier() : Boolean {
```

```
  return ClassObject.allInstances().select(c|c.identifier = self).size() = 1;
 }

 operation String isRecognisedIdentifier() : Boolean {
  if (’#’.isSubstringOf(self)) {
   return self.canLocateExternalModel() and self.canLocateExternalModelElement();

  } else {
   return self.isUniqueIdentifier();
  }
 }

 operation String getMessage() : String {
  if (’#’.isSubstringOf(self)) {
   if (not self.canLocateExternalModel()) {
    return ’Model not found: ’ + self.split(’#’).first();
   }

   return ’Model element not found: ’ + self.split(’#’).last();

  } else {
   return ’Unrecognised identifier: ’ + self;
  }
 }

 operation String canLocateExternalModel() : Boolean {
  return EmfTool.resourceExists(self.split(’#’).first());
 }

 operation String canLocateExternalModelElement() : Boolean {
  return EmfTool.modelElementExists(self);
 }

 operation String toClass() : Metamodel!EClass {
  if (’#’.isSubstringOf(self)) {
   -- External object reference, locate in external model

   var object := EmfTool.getEObject(self);

   if (object.isDefined()) {
    return object.eClass();
   }

  } else {
   -- Internal object reference, located in current model

```

```
      var object := ClassObject.allInstances().selectOne(c|c.identifier = self);

      if (object.isDefined()) {
        return object.toClass();
      }
    }
}

operation ClassObject toClass() : Metamodel!EClass {
  return self.getEClass();
}

operation Metamodel!EClass getAllClassObjects() : Collection(ClassObject) {
  return ClassObject.all.select(c|c.hasEClass() and c.getEClass().name = self.
      name);
}

operation Metamodel!EClass getAllFeaturesThatRequireAValue() : Sequence(
    EReference) {
  return self.eAllStructuralFeatures.select(f|f.lowerBound > 0 and f.changeable
      and not f.transient);
}

operation Metamodel!EClass getAllReferencesThatRequireAValue() : Sequence(
    EReference) {
  return self.eAllReferences.select(f|f.lowerBound > 0 and f.changeable and not f
      .transient);
}

operation Metamodel!EReference hasOppositeReferencing(classObject : ClassObject)
      : Boolean {
  if (self.eOpposite.isDefined()) {
    for (class in self.eType.getAllClassObjects()) {
      var slot := class.findSlot(self.eOpposite.name);

      if (slot.isDefined()) {
        if (slot.isKindOf(ReferenceSlot) and slot.values.includes(classObject.
            identifier)) {
          return true;

        } else if (slot.classObjects.includes(classObject)) {
          return true;
        }

      }
    }
  }
```

```

 return false;
}

operation ClassObject getAllReferencesThatRequireAValueButDontHaveOne() :
   Sequence(EReference) {
  return self.toClass().getAllReferencesThatRequireAValue().reject(r|self.slots
     .exists(s|s.feature = r.name) or r.hasOppositeReferencing(self));
}

operation Metamodel!EReference aValueCanBeInferred() : Boolean {
 return self.isContainment and self.eType.getAllFeaturesThatRequireAValue().
    isEmpty();
}


operation Sequence toString() : String {
 var result : String := '';

 for (element in self) {
  result := result + element.toString();
  if (hasMore) { result := result + ', '; }
 }

 return result;
}
```

Listing A.4: Syntactic and Conformance Constraints in EVL.

### A.2.3 Intermediate Model to Target Model

The EGL template in Listing A.5 is used to generate an ETL transformation between intermediate models and target models.

```
pre {
 var EmfTool := new Native('org.eclipse.epsilon.emc.emf.tools.EmfTool');
}

[% if (not PackageObject.all.isEmpty()) { %]
 [% for (class in getAllEClassesUsed()) { %]
 rule Object2[%=class.name%]
  transform o : Intermediate!ClassObject
  to t : Model!`[%=class.name%]` {

  guard: o.type = '[%=class.name%]'

 [% for (attribute in class.eAllAttributes) { %]
  if (o.findSlot('[%=attribute.name%]').isDefined()) {
```

```
   [% if (attribute.isMany()) { %]
    for (value in o.findSlot(’[%=attribute.name%]’).values) {
     t.`[%=attribute.name%]`.add(value);

      t.`[%=attribute.name%]`.add(value);

    }
   [% } else { %]

     t.`[%=attribute.name%]` := o.findSlot(’[%=attribute.name%]’).values.first;
   [% } %]
  }
 [% } %]

 [% for (reference in class.eAllReferences) { %]
  if (o.findSlot(’[%=reference.name%]’).isDefined()) {
   [% if (reference.isMany()) { %]
    for (object in o.findSlot(’[%=reference.name%]’).getEObjects()) {
     t.`[%=reference.name%]`.add(object);
    }
   [% } else { %]
    t.`[%=reference.name%]` := o.findSlot(’[%=reference.name%]’).getEObjects().first();
   [% } %]
  }
 [% } %]
 }
 [% } %]
[% } %]

operation ReferenceSlot getEObjects() : Sequence {
 return self.values.collect(i:String | i.getEObject(self));
}

operation ContainmentSlot getEObjects() : Sequence {
 return self.classObjects.collect(o:Intermediate!ClassObject | o.equivalent());
}

operation String getEObject(slot : ReferenceSlot) : Any {
 if (’#’.isSubstringOf(self)) {
  -- External object reference, locate in external model
  return EmfTool.getEObject(self);

 } else {
  -- Internal object reference, located in current model
  return slot.getClassObjects().selectOne(c|c.identifier = self).equivalent();
```

```
  }
}


[%
 operation getAllEClassesUsed() : Sequence {
 var types := new Set;
 var classes := new Sequence;

 for (classObject in ClassObject.all) {
   if (types.excludes(classObject.type)) {
     types.add(classObject.type);
     classes.add(classObject.getEClass());
   }
 }

 return classes;
 }
%]
```

Listing A.5: EGL template that generates an intermediate model to target model transformation (in ETL)

# Appendix B

# A Graphical Editor for Process-Oriented Programs

This appendix describes the design and implementation of a prototypical graphical editor for process-oriented programs. The work presented here was conducted in collaboration with Adam Sampson, then a Research Associate at the University of Kent. The way in which the graphical editor changed throughout its development provided was used for the evaluation presented in Section 6.1.

The purpose of the collaboration was to explore the suitability of MDE for designing a graphical notation and editor for programs written in process-oriented programming languages, such as occam-$\pi$ [Welch & Barnes 2005]. The collaboration produced a prototypical graphical editor implemented with EMF and GMF (Section 2.3).

Process-oriented programs are specified in terms of three core concepts: processes, connection points and channels. Processes are the fundamental building blocks of a process-oriented program. Channels are the mechanism by which processes communicate, and are unidirectional. Connection points define the channels on which a process can communicate. Connection points are used to specify the way in which a process can communicate, and can optionally be bound to a channel. Because channels are unidirectional, connection points are either reading (consume messages from the channel) or writing (generate messages on the channel).

The graphical notation and editor were implemented in an iterative and incremental manner. The abstract syntax of the domain was specified as a metamodel, captured in Ecore, which is the metamodelling language provided by EMF. The graphical concrete syntax was specified with GMF, using EuGENia [Kolovos *et al.* 2010]. EMF and GMF are described more thoroughly in Section 2.3.

The remainder of this appendix describes the six iterations that took place during the development of the graphical editor for process-oriented programs. Each section describes the goal of the iteration, the changes made to the metamodel to meet the goal, and the impact of the changes on models that had been constructed in previous iterations. The way in which models were migrated with a user-driven co-evolution approach is also described.

## B.1 Iteration 1: Processes and Channels

Development began by identifying two key concepts for modelling process-oriented programs. From examples of process-oriented programs, process and channel were identified as the most important concepts, and consequently the metamodel shown in Figure B.1 was constructed.

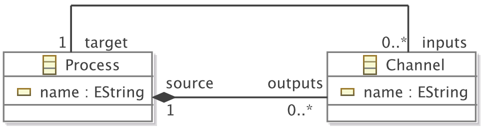


Figure B.1: The process-oriented metamodel after one iteration.

Additionally, a graphical concrete syntax was chosen for processes and channels. The former were represented as boxes, and the latter as lines. EuGENia annotations were added to the metamodel, resulting in the metamodel shown in Listing B.1. Line 1 of Listing B.1 uses the "@gmf.node" EuGENia annotation to indicate that processes are to be represented as boxes with a label equal to the value of the `name` feature. Line 9 uses the "@gmf.link" EuGENia annotation to indicate that channels are to be represented as lines between `source` and `target` processes with a label equal to the value of the `name` feature.

```
 1  @gmf.node(label="name")
 2  class Process {
 3    attr String name;
 4
 5    ref Channel[*]#target inputs;
 6    val Channel[*]#source outputs;
 7  }
 8
 9  @gmf.link(source="source", target="target", label="name")
10  class Channel {
11    attr String name;
12    ref Process[1]#outputs source;
13    ref Process[1]#inputs target;
14  }
```

Listing B.1: The annotated process-oriented metamodel after one iteration

To generate code for the graphical editor, EuGENia was invoked on the annotated metamodel shown in Listing B.1. However, EuGENia failed with an error, because no "root" element had been specified. GMF, the graphical modelling framework used by EuGENia, requires one metaclass (termed the root) to be specified as a container for all diagram elements. The root metaclass cannot be a GMF node or a link, and so the second iteration involved adding an additional metaclass for interoperability with GMF.

## B.2 Iteration 2: Interoperability with GMF

In the second iteration, an additional metaclass, `Model`, was added to the metamodel as shown in Figure B.2. The `Model` metaclass was used to provide GMF with a container for storing all of the diagram elements for each process-oriented diagram.

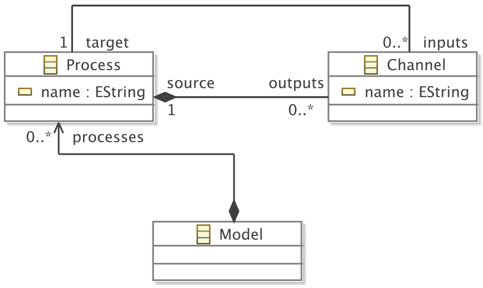


Figure B.2: The process-oriented metamodel after two iterations.

The `Model` metaclass was annotated with “@gmf.diagram” to indicate that it should be used as the diagram’s root element (Listing B.2). Root elements do not have a concrete syntax and do not appear in the graphical editor.

EuGENia was invoked on the annotated metamodel shown in Listing B.2 to produce code for the graphical editor. Figure B.3 shows a model that was constructed to test the generated editor and comprised two processes, `P1` and `P2`, and one channel, `a`.

## B.3 Iteration 3: Shared Channels

In previous iterations, channels had been contained within their source process. The nested structure made it more difficult to explore process-oriented models in EMF’s tree editor due to the additional level of nesting. Consequently, the metamodel was changed such that channels were contained in the root element, rather than in the source process, resulting in the metamodel shown in Figure B.4.

No additional EuGENia annotations were added to the metamodel during this iteration. In other words, the graphical notation (concrete syntax) was not changed, and the resulting editor was identical in appearance to the previous one. However, the EMF tree editor showed just one level of nesting (everything is contained inside model).

The existing models required migration because of the way in which XMI differentiates between reference and containment values. Each channel was moved to the new `channels` reference of `Model`, and existing values in the `outputs` reference of `Conn-`

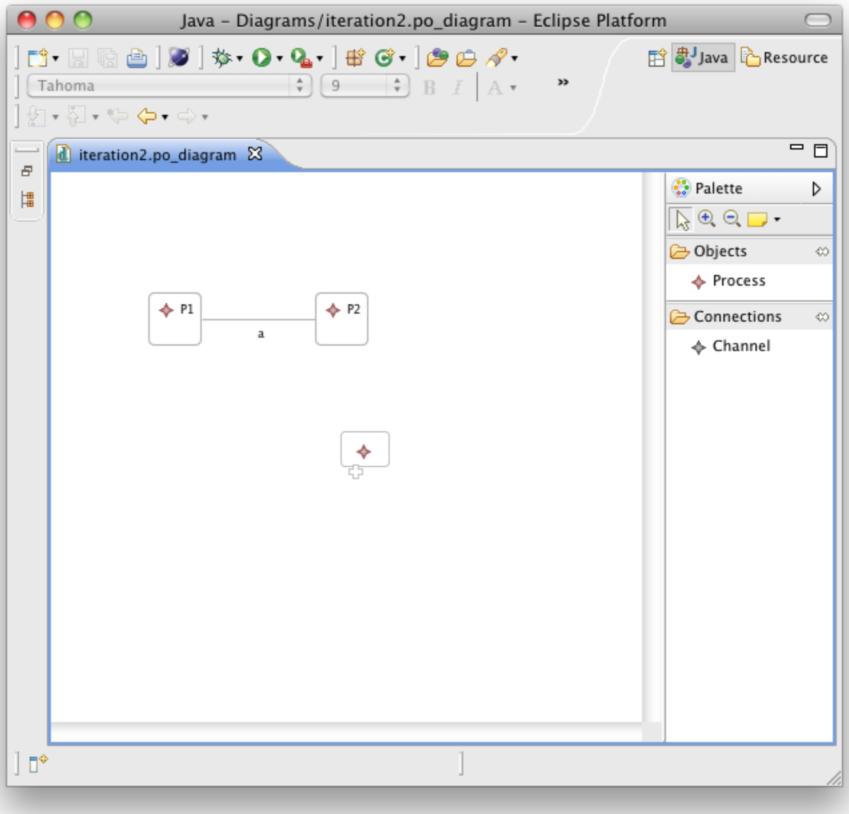


Figure B.3: A diagram after the second iteration.

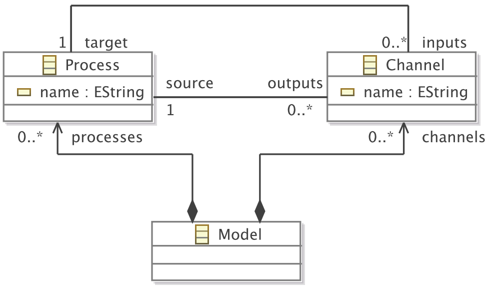
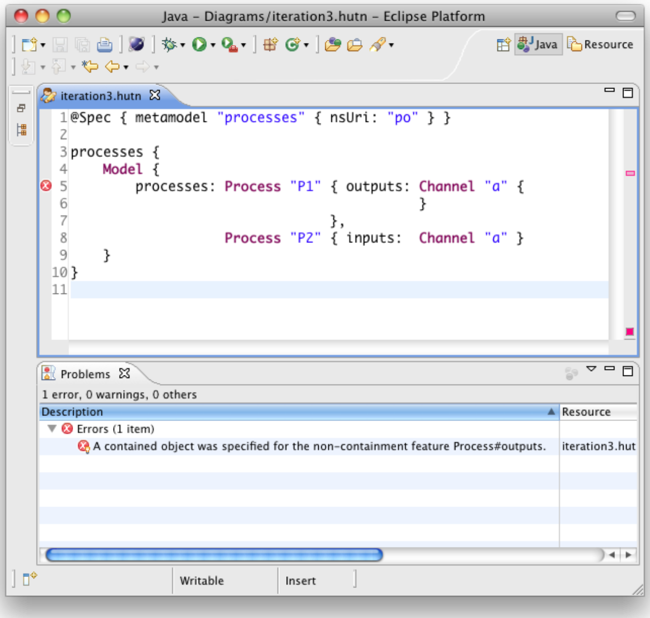


Figure B.4: The process-oriented metamodel after three iterations.

```
@gmf.diagram
class Model {
  val Process[*] processes;
}

@gmf.node(label="name")
class Process {
  attr String name;

  ref Channel[*]#target inputs;
  val Channel[*]#source outputs;
}


@gmf.link(source="source", target="target", label="name")
class Channel {
  attr String name;
  ref Process[1]#outputs source;
  ref Process[1]#inputs target;
}
```

Listing B.2: The annotated process-oriented metamodel after two iterations

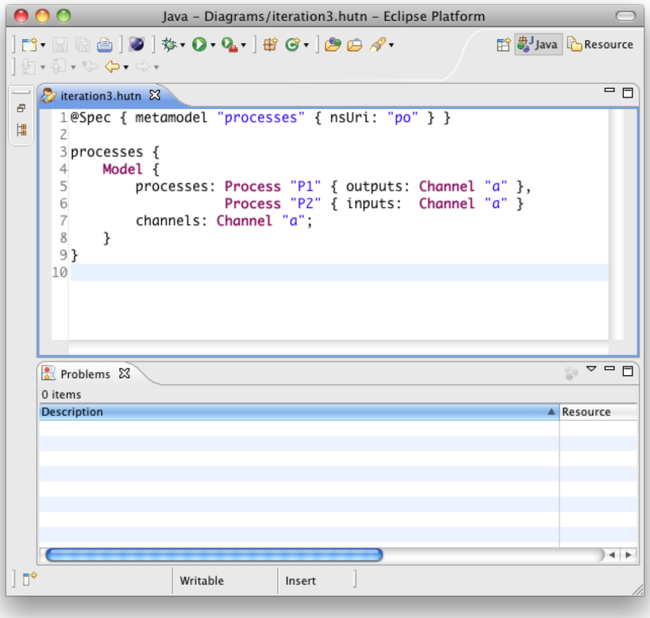
`ectionPoint` were changed to a reference value. Figure B.5(a) shows the HUTN for a model prior to migration, and Figure B.5(b) shows the reconciled, migrated HUTN.



```
@Spec { metamodel "processes" { nsUri: "po" } }

processes {
    Model {
        processes: Process "P1" { outputs: Channel "a" {
                                           }
                                },
                   Process "P2" { inputs:  Channel "a" }
    }
}

```

```
1 error, 0 warnings, 0 others
Description                                                                       Resource
▼ Errors (1 item)
    A contained object was specified for the non-containment feature Process#outputs.   iteration3.hut
```

(a) HUTN prior to migration



```
@Spec { metamodel "processes" { nsUri: "po" } }

processes {
    Model {
        processes: Process "P1" { outputs: Channel "a" },
                   Process "P2" { inputs:  Channel "a" }
        channels: Channel "a";
    }
}

```

```
0 items
Description                                                                       Resource
```

(b) HUTN after migration

Figure B.5: Migration between the second and third versions of the process-oriented metamodel, with Epsilon HUTN (Section 5.2)

## B.4 Iteration 4: Connection Points

The fourth iteration involved capturing a third domain concept, connection points, in the graphical notation. When a process is specified, the ways in which it can communicate are declared as connection points. When a process is instantiated, channels are connected to its connection points, and messages flow in and out of the process. The graphical notation was to be used to describe both instantiated processes and types of process, the metamodel was changed to model connection points.

The iteration resulted in the metamodel shown in Figure B.6. `ConnectionPoint` was introduced as an association class for the references between `Process` and `Channel`.

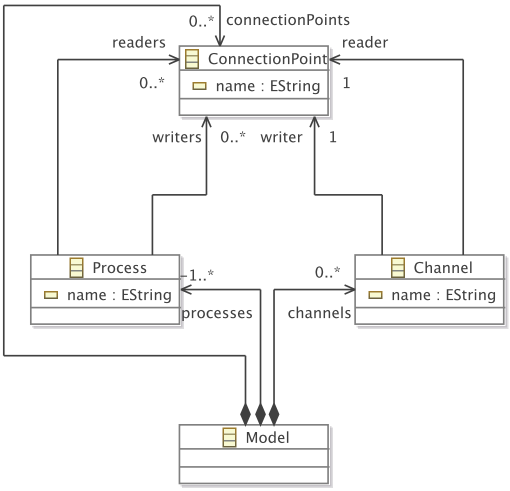


Figure B.6: The process-oriented metamodel after the fourth iteration.

To specify concrete syntax for connection points, additional EuGENia annotations were added to the metamodel as shown in Listing B.3. The `ConnectionPoint` class was annotated with a “@gmf.node” to specify that connections points were to be represented as circles, labelled with the value of the `name` attribute. The circles were to be affixed to the boxes used to represent processes, and, hence, “@gmf.affixed” annotations are used on lines 12 and 15.

```
1  @gmf.diagram
2  class Model {
3     val Process[*] processes;
4     val Channel[*] channels;
5     val ConnectionPoint[*] connectionPoints;
6  }
7
8  @gmf.node(label="name")
9  class Process {
```

```
  attr String name;

  @gmf.affixed
  ref ConnectionPoint[*] readers;

  @gmf.affixed
  ref ConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="true")
class Channel {
 attr String name;
  ref ConnectionPoint[1] reader;
  ref ConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false", figure="
    ellipse", size="15,15")
class ConnectionPoint {
  attr String name;
}
```

Listing B.3: The annotated process-oriented metamodel after four iterations

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. A larger test model was constructed to test the editor, and is shown in Figure B.7. The existing models required migration because the `inputs` and `outputs` references of `Process` and the `source` and `target` references of `Channel` had been removed.

To migrate each existing model, two connection points were created for each channel in the model. The `source` and `target` reference of the channel was changed to reference the new connection points, as were the corresponding values of the `readers` and `writers` references of the relevant processes. Figure B.8(a) shows the HUTN for a model prior to migration, and Figure B.8(b) shows the reconciled, migrated HUTN.

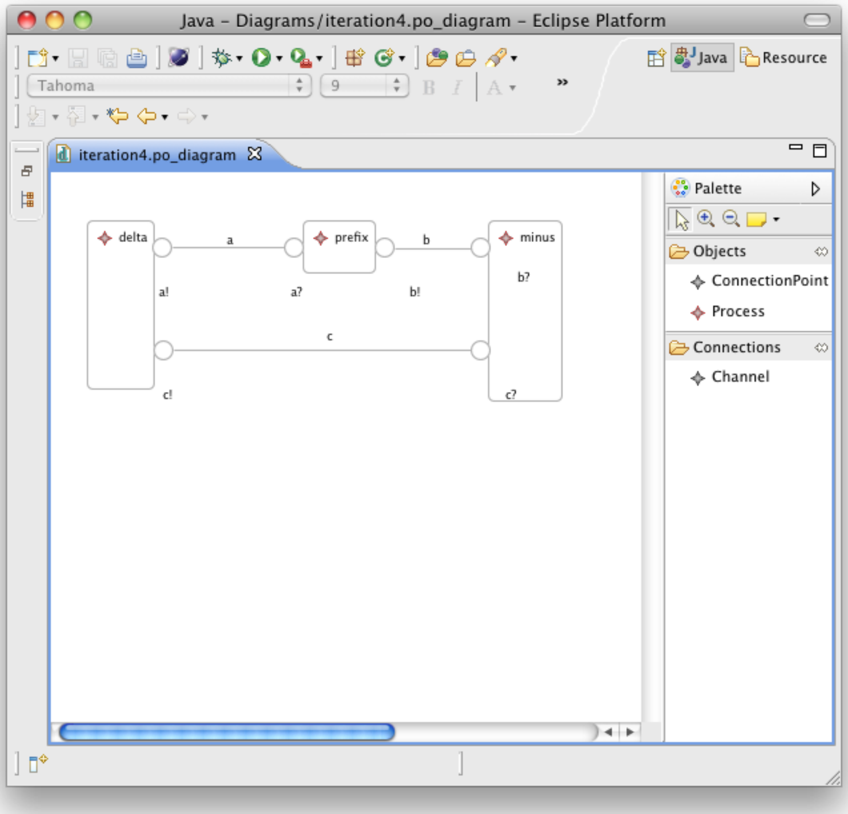
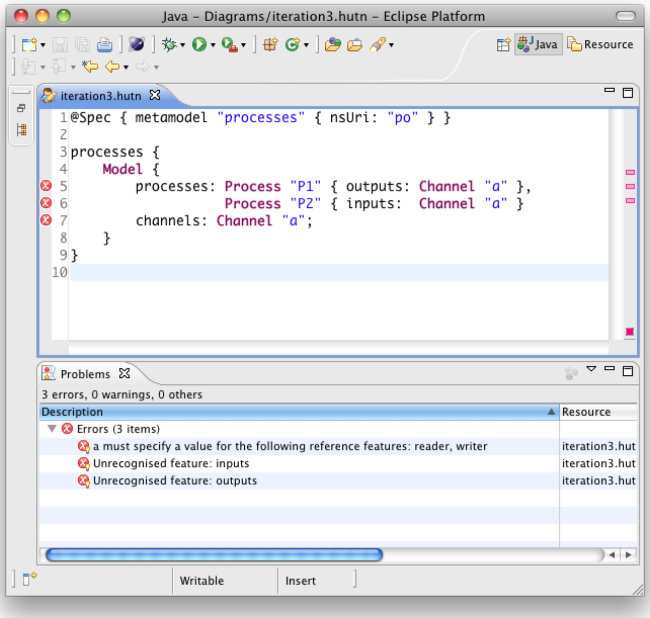


Figure B.7: A diagram after the fourth iteration.

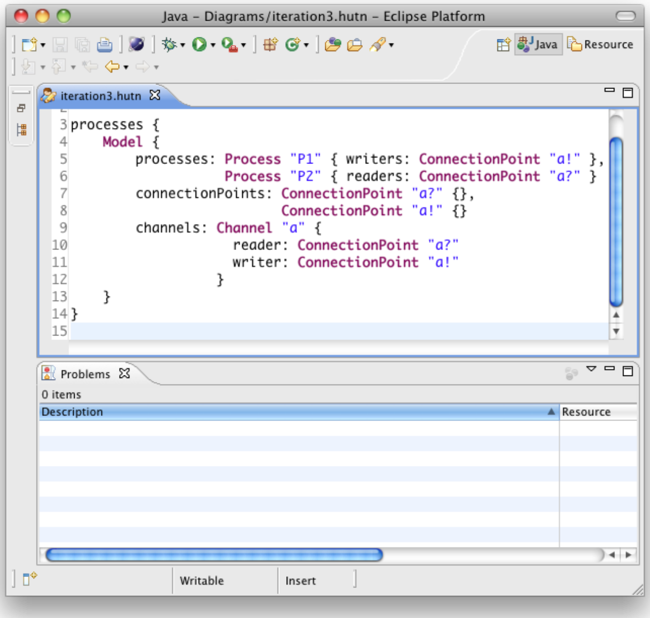


```
@Spec { metamodel "processes" { nsUri: "po" } }

processes {
    Model {
        processes: Process "P1" { outputs: Channel "a" },
                   Process "P2" { inputs:  Channel "a" }
        channels: Channel "a";
    }
}
```

3 errors, 0 warnings, 0 others

| Description | Resource |
|---|---|
| Errors (3 items) | |
| a must specify a value for the following reference features: reader, writer | iteration3.hut |
| Unrecognised feature: inputs | iteration3.hut |
| Unrecognised feature: outputs | iteration3.hut |

(a) HUTN prior to migration



```
processes {
    Model {
        processes: Process "P1" { writers: ConnectionPoint "a!" },
                   Process "P2" { readers: ConnectionPoint "a?" }
        connectionPoints: ConnectionPoint "a?" {},
                          ConnectionPoint "a!" {}
        channels: Channel "a" {
                      reader: ConnectionPoint "a?"
                      writer: ConnectionPoint "a!"
                  }
    }
}
```

0 items

(b) HUTN after migration

Figure B.8: Migration between the third and fourth versions of the process-oriented metamodel, with Epsilon HUTN (Section 5.2)

## B.5 Iteration 5: Connection Point Types

Channels are unidirectional, and so connection points are either *reading* or *writing*. A process uses the former to consume messages from a channel, and the latter to produce messages on a channel. Testing the graphical editor produced in the fourth iteration showed that it was not immediately obvious as to which connection points were reading and which were writing. The fifth iteration involved changing the graphical editor to better distinguish between reading and writing connection points.

The iteration resulted in the metamodel shown in Figure B.9. `ConnectionPoint` was made abstract, and two subclass, `ReadingConnectionPoint` and `WritingConnectionPoint`, were introduced. The four references to `ConnectionPoint` were changed to reference one of the two subclasses.

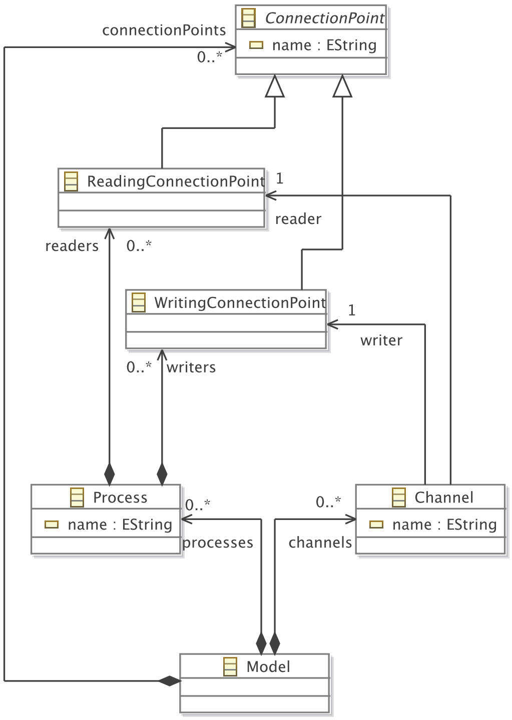


Figure B.9: The process-oriented metamodel after five iterations.

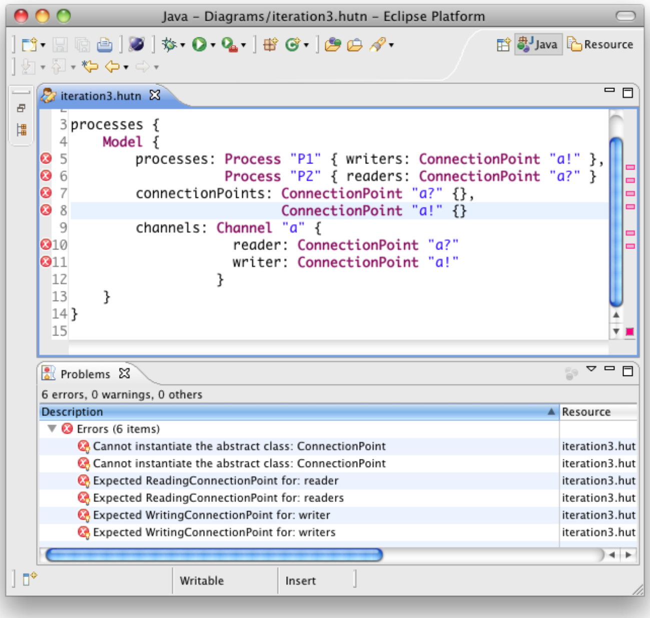
The graphical notation was changed, as shown in Listing B.4. The `WritingConnectionPoint` class was annotated with an additional colour attribute to specify that writing connection points were to be represented with a black circle. White is the default colour for a “@gmf.node” annotation, and so reading connection points were represented as white circles.

```
@gmf.diagram
class Model {
```

```
  val Process[*] processes;
  val Channel[*] channels;
  val ConnectionPoint[*] connectionPoints;
}

@gmf.node(label="name")
class Process {
  attr String name;

  @gmf.affixed
  ref ReadingConnectionPoint[*] readers;

  @gmf.affixed
  ref WritingConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="true")
class Channel {
 attr String name;
  ref ReadingConnectionPoint[1] reader;
  ref WritingConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false", figure="
    ellipse", size="15,15")
abstract class ConnectionPoint {
  attr String name;
}

class ReadingConnectionPoint extends ConnectionPoint {}

@gmf.node(color="0,0,0")
class WritingConnectionPoint extends ConnectionPoint {}
```

Listing B.4: The annotated process-oriented metamodel after five iterations

A new version of the graphical editor was generated by invoking EuGENia on the annotated metamodel. All of the existing models required migration, because `ConnectionPoint` was now an abstract class, and could no longer be instantiated. Section 6.1 describes the way in which models were migrated after the changes made during this iteration. Briefly, migration involved replacing every instantiation of `ConnectionPoint` with an instantiation of either `ReadingConnectionPoint` or `WritingConnectionPoint`. The former was used when a connection point was used as the value of a channel's `reader` feature and the latter when when a connection point was used as the value of a channel's `writer` feature. Figure B.10(a) shows the HUTN for a model prior to migration, and Figure B.10(b) shows the reconciled, migrated HUTN.

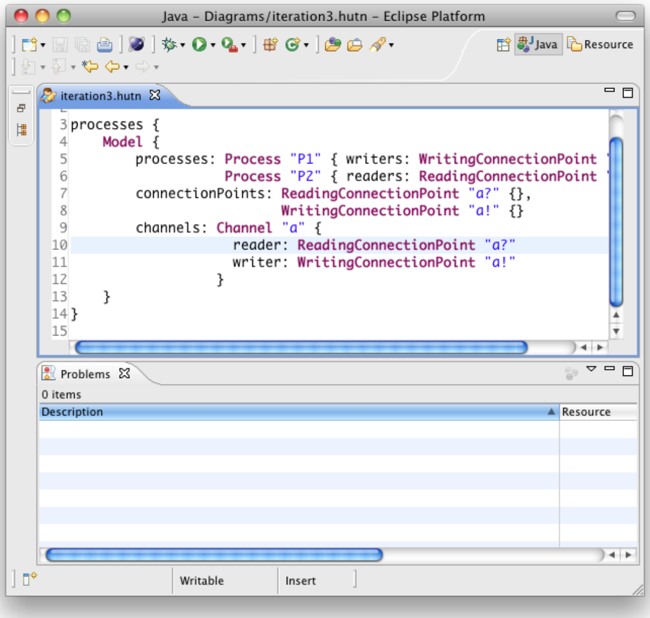


```
processes {
    Model {
        processes: Process "P1" { writers: ConnectionPoint "a!" },
                   Process "P2" { readers: ConnectionPoint "a?" }
        connectionPoints: ConnectionPoint "a?" {},
                          ConnectionPoint "a!" {}
        channels: Channel "a" {
                      reader: ConnectionPoint "a?"
                      writer: ConnectionPoint "a!"
                  }
    }
}
```

6 errors, 0 warnings, 0 others

| Description | Resource |
|---|---|
| Errors (6 items) | |
| Cannot instantiate the abstract class: ConnectionPoint | iteration3.hut |
| Cannot instantiate the abstract class: ConnectionPoint | iteration3.hut |
| Expected ReadingConnectionPoint for: reader | iteration3.hut |
| Expected ReadingConnectionPoint for: readers | iteration3.hut |
| Expected WritingConnectionPoint for: writer | iteration3.hut |
| Expected WritingConnectionPoint for: writers | iteration3.hut |

(a) HUTN prior to migration



```
processes {
    Model {
        processes: Process "P1" { writers: WritingConnectionPoint 
                   Process "P2" { readers: ReadingConnectionPoint 
        connectionPoints: ReadingConnectionPoint "a?" {},
                          WritingConnectionPoint "a!" {}
        channels: Channel "a" {
                      reader: ReadingConnectionPoint "a?"
                      writer: WritingConnectionPoint "a!"
                  }
    }
}
```

0 items

| Description | Resource |
|---|---|

(b) HUTN after migration

Figure B.10: Migration between the fourth and fifth versions of the process-oriented metamodel, with Epsilon HUTN (Section 5.2)

## B.6 Iteration 6: Nested Processes and Channels

The final iteration involved changing the graphical editor such that processes and channels could be nested inside other processes. In some process-oriented languages, such as occam-$\pi$ [Welch & Barnes 2005], processes can be specified in terms of other, internal processes.

To support the decomposition of processes into other processes and channels, the `nestedProcess` and `nestedChannel` references were added to the `Process` class, as shown in Figure B.11.

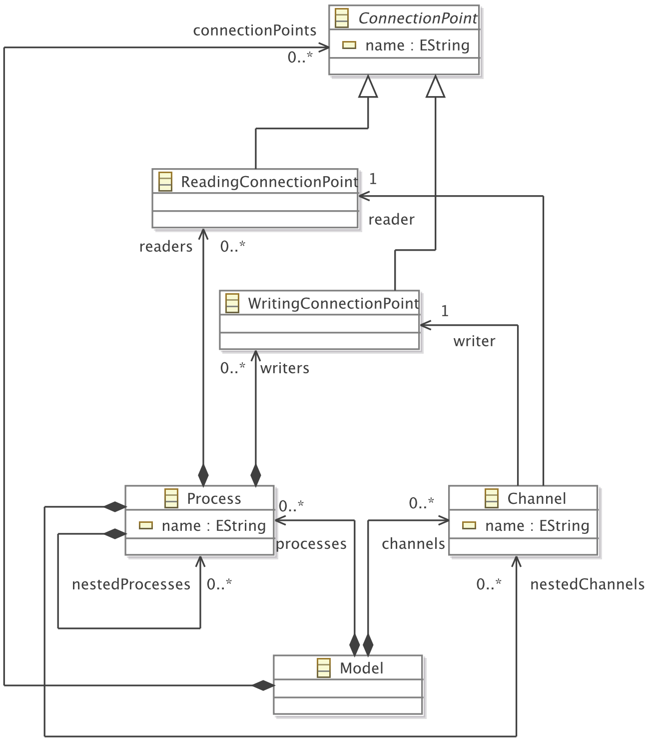


Figure B.11: The process-oriented metamodel after six iterations.

As shown in Listing B.5, the "@gmf.compartment" annotation was added to the `nestedProcess` to indicate that processes can be placed inside other processes in the graphical editor.

```
@gmf.diagram
class Model {
  val Process[*] processes;
  val Channel[*] channels;
  val ConnectionPoint[*] connectionPoints;
}
```

```
@gmf.node(label="name")
class Process {
  attr String name;

  @gmf.compartment
  val Process[*] nestedProcesses;
  val Channel[*] nestedChannels;

  @gmf.affixed
  ref ReadingConnectionPoint[*] readers;

  @gmf.affixed
  ref WritingConnectionPoint[*] writers;
}


@gmf.link(source="reader", target="writer", label="name", incoming="true")
class Channel {
 attr String name;
  ref ReadingConnectionPoint[1] reader;
  ref WritingConnectionPoint[1] writer;
}

@gmf.node(label="name", label.placement="external", label.icon="false", figure="
    ellipse", size="15,15")
abstract class ConnectionPoint {
  attr String name;
}

class ReadingConnectionPoint extends ConnectionPoint {}

@gmf.node(color="0,0,0")
class WritingConnectionPoint extends ConnectionPoint {}
```

Listing B.5: The annotated process-oriented metamodel after six iterations

EuGENia was invoked on the annotated metamodel to produce the final version of the graphical editor. An additional model was constructed to check the nesting of processes, and is shown in Figure B.12. Because the changes made to the metamodel in this iteration involved only adding new features, no migration of existing models was necessary.

## B.7 Summary

This appendix has described the way in which a graphical editor for process-oriented programs was designed and implemented using an iterative style of development. A metamodel was used to capture the key concepts of the domain, and to generate code

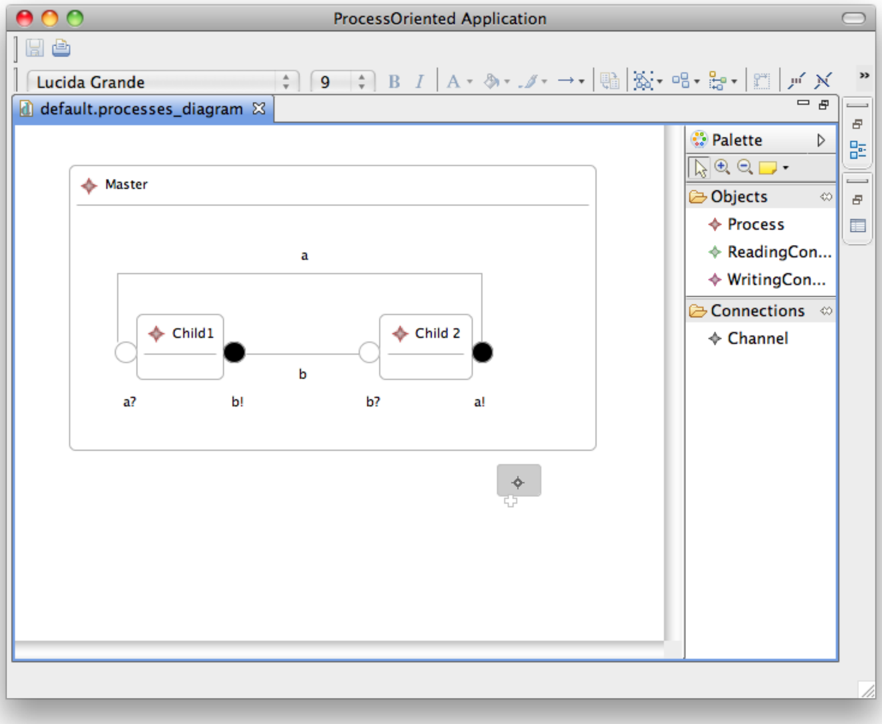


Figure B.12: A diagram after the final iteration.

for a graphical editor. Each iteration involved changing the metamodel either to correct unintended behaviour in the editor (iterations 3 and 5), to facilitate interoperability with other tools (iteration 2) or to add new features (iterations 1, 4 and 6). The metamodel changes described in the fifth iteration are used for evaluation of the thesis research in Section 6.1.

# Appendix C

# Co-evolution Examples

This appendix describes the co-evolution examples used for evaluation in Chapter 6. The examples were taken from real-world MDE projects and are distinct from the examples used for analysis in Chapter 4.

Below, each section details examples from one project, describes metamodel changes and presents model migration strategies. Each model migration strategy is presented in the three model migration languages used for evaluating conservative copy in Section 6.2, and lines that contain *a model operation* (a statement that changes the migrated model) are highlighted. Section 6.2 describes model operations and the three model migration languages in more detail.

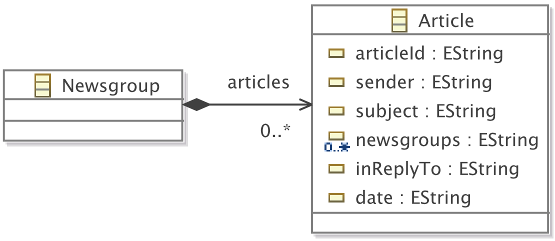
## C.1 Newsgroups Examples

The first set of examples were taken from a project that performed statistical analysis of NNTP newsgroups, developed by Dimitris Kolovos, a lecturer in this department. The analysis was implemented using a metamodel to capture domain-specific concepts, a text-to-model transformation for parsing newsgroup messages, and a model-to-model transformation for recording the results of the analysis.

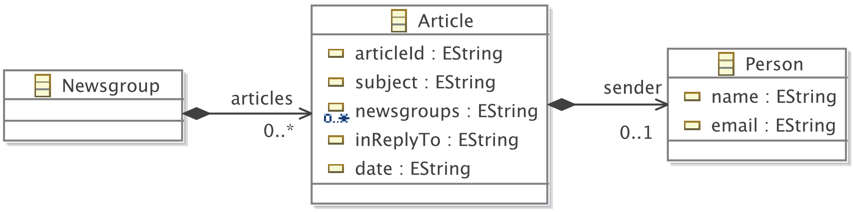
The metamodel and transformations were developed in an iterative and incremental manner. Five iterations of the metamodel and transformations were made available by Kolovos, two of which involved metamodel changes that affected the conformance of existing models and are described below. In the other three iterations, the metamodel changes were additive, did not lead to model migration, and are not described here.

### C.1.1 Extract Person

At the start of the second iteration, the newsgroups metamodel, show in Figure C.1(a), captured two domain concepts, newsgroups and articles. The iteration involved separating the domain concepts of authors and articles. At the start of the iteration, the `Article` class defined a string attribute called `sender` as shown in Figure C.1(a). To make it easier to recognise when several articles were written by the same person, the



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.1: Newsgroups metamodel during the Extract Person iteration

`Person` class was introduced, and the `sender` attribute was replaced with a reference to the `Person` class as shown in Figure C.1(b).

Existing models were migrated by deriving a `Person` object from the `sender` feature of each `Article`. The values of the `sender` feature used one of two forms: `abc@example.com (Full Name)` or `"Full Name" abc@example.com`.

Listings C.1, C.2 and C.3 show the model migration strategy in ATL, COPE and Flock respectively. The `toEmail()` and `toName()` operations are used to extract names and email addresses, are defined without using any model operations, and are omitted from the listings below.

```
module ExtractPerson;

create Migrated : After from Original : Before;

rule Newsgroups {
  from
    o : Before!Newsgroup
  to
    m : After!Newsgroup (
      articles <- o.articles
    )
}

rule Articles {
  from
```

```
    o : Before!Article
  to
    m : After!Article (
      articleId <- o.articleId,
      subject  <- o.subject,
      newsgroups <- o.newsgroups,
      inReplyTo <- o.inReplyTo,
      date     <- o.date,
      sender   <- p
    ),
    p : After!Person (
      name <- o.sender.toName(),
      email <- o.sender.toEmail()
    )
}
```

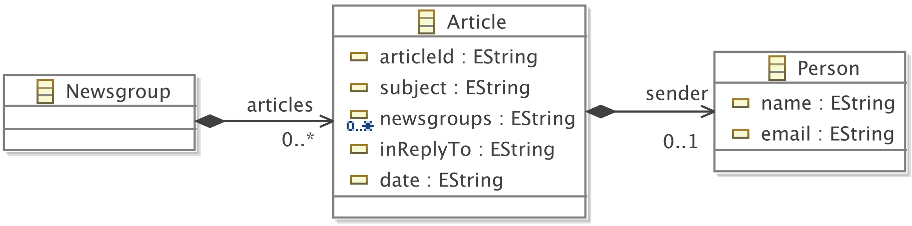
Listing C.1: The Newsgroup Extract Person migration in ATL

```
toPerson = { str ->
  def person = personClass.newInstance();
  person.email = str.toEmail()
  person.name = str.toName()
  return person
}

for (article in extractperson.Article.allInstances) {
  def sender = article.unset(sender)
  article.sender = toPerson(sender)
}
```

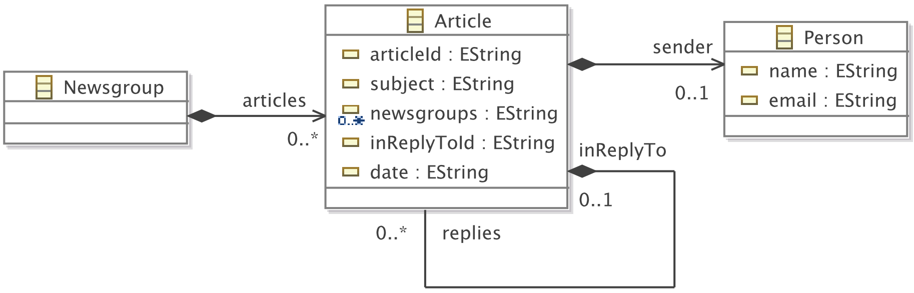
Listing C.2: The Newsgroup Extract Person migration in Groovy-for-COPE

```
migrate Article {
  migrated.sender := original.sender.toPerson();
}

operation String toPerson() : Migrated!Person {
  var person := new Migrated!Person;
  person.name := self.toName();
  person.email := self.toEmail();
  return person;
}
```

Listing C.3: The Newsgroup Extract Person migration in Flock



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.2: Newsgroups metamodel during the Resolve Replies iteration

### C.1.2 Resolve Replies

The Resolve Replies iteration made explicit the lineage of each article by moving replies to an article such that they were contained in the original article. At the start of the iteration (Figure C.2(a)), each `Article` was assigned a unique identifier in the `articleId` feature. The `inReplyTo` feature was specified for `Articles` written in reply to others. At the end of the iteration, the `inReplyTo` attribute was replaced with a reference of type `Article`. The `inReplyTo` attribute was renamed to `inReplyToId` (and, in a future iteration, was removed from the metamodel).

Listings C.4, C.5 and C.6 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved dereferencing the `inReplyTo` value to determine a parent `Article`, and then setting the `inReplyTo` reference to the parent `Article`.

```
module ResolveReplies;

create Migrated : After from Original : Before;

rule Newsgroups {
  from
    o : Before!Newsgroup
  to
```

```
   m : After!Newsgroup (
     articles <- o.articles
   )
}

rule Articles {
 from
   o : Before!Article
 to
   m : After!Article (
     articleId <- o.articleId,
     subject  <- o.subject,
     newsgroups <- o.newsgroups,
     inReplyToId <- o.inReplyTo,
     date     <- o.date,
     sender   <- o.sender
   )
 do {
   if (not o.inReplyTo.oclIsUndefined() and After!Article.allInstances()->exists
       (a|a.articleId = o.inReplyTo)) {
     After!Article.allInstances()->select(a|a.articleId = o.inReplyTo)->first().
     replies <- m;
   }
 }
}
```

Listing C.4: The Newsgroup Resolve Replies migration in ATL

```
for (article in extractperson.Article.allInstances) {
 def replyToId = article.unset(replyTo)
 article.replyToId = replyToId
 article.replyTo = Article.allInstances.find { it.articledId = article.replyToId
      }
}
```

Listing C.5: The Newsgroup Resolve Replies migration in Groovy-for-COPE

```
migrate Article {
 migrated.inReplyToId := original.inReplyTo;
 migrated.inReplyTo := Migrated!Article.all.selectOne(a|a.articleId = migrated.
    inReplyToId);
}
```

Listing C.6: The Newsgroup Resolve Replies migration in Flock

## C.2 UML Example

This section describes the co-evolution example taken from the evolution of the Unified Modeling Language (UML) between versions 1.4 [OMG 2001] and 2.2 [OMG 2007b]. Activity diagrams, in particular, changed radically between UML versions 1.4 and 2.2. In the former, activities were defined as a special case of state machines, while in the latter they were defined with a more general semantic base[1] [Selic 2005].

The UML 1.4 and 2.2 specifications are defined in different metamodelling languages. The former uses XMI 1.4 and the latter XMI 2.2. Of the co-evolution tools discussed in this thesis, only Epsilon Flock interoperates with XMI 1.4. To enable the use of other co-evolution tools with the UML metamodel changes, the author reconstructed part of the UML 1.4 metamodel in XMI 2.2.

The migration semantics were identified by comparing the UML 1.4 and UML 2.2 specifications, and by discussing the metamodel evolution with other UML experts. As described in Section 6.4, the UML 2.2 specification appears to be ambiguous with respect to the way in which UML 1.4 `ObjectFlowStates` should be migrated to conform to the UML 2.2 metamodel. The migration strategies presented here assume the semantics of the core task described in Section 6.4: `ObjectFlowStates` are replaced with `ObjectNodes`.

### C.2.1 Activity Diagrams

Figures C.3(a) and C.3(b) are simplifications of the activity diagram metamodels from versions 1.4 and 2.2 of the UML specification, respectively. In the interest of clarity, some features and abstract classes have been removed from Figures C.3(a) and C.3(b).

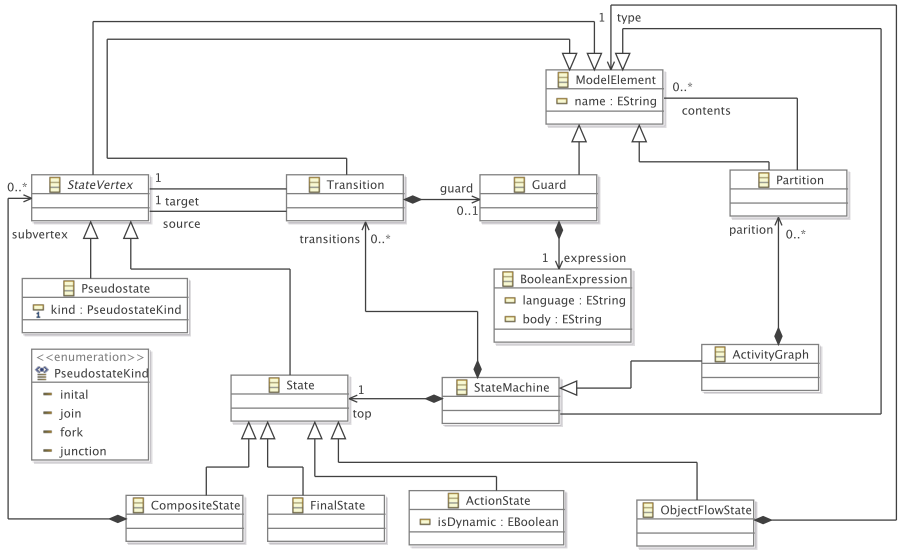
Some differences between Figures C.3(a) and C.3(b) are: activities have been changed such that they comprise nodes and edges, actions replace states in UML 2.2, and the subtypes of control node replace pseudostates.

Listings C.7, C.8 and C.9 show the model migration strategy in ATL, COPE and Flock respectively. Migration mostly involved restructuring data by storing values in features of a different name, and retyping `Pseudeostates`.

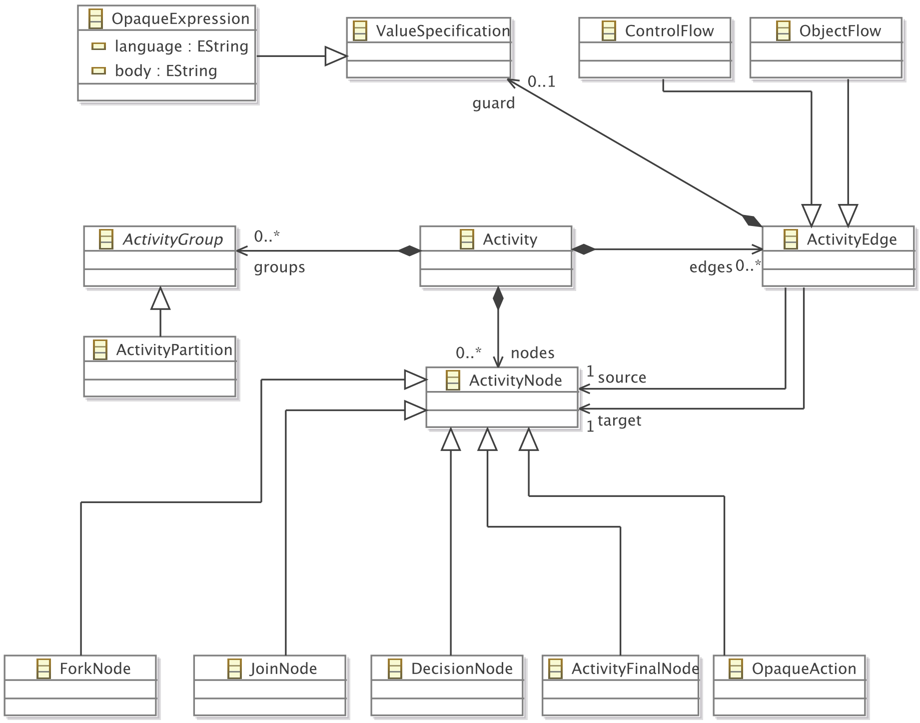
```
module ActivityGraph;

create Migrated : After from Original : Before;

rule ActivityGraph {
 from
   o : Before!ActivityGraph
 to
   p : After!Package (
     packagedElement <- m
   ),
   m : After!Activity (
```

---

[1] A variant of generalised coloured Petri nets.



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.3: Activities in UML 1.4 and UML 2.2

```
      name <- o.name,
      node <- o.top.subvertex,
      edge <- o.transitions,
      group <- o.partition
    )
}

rule Partitions {
  from
    o : Before!Partition
  to
    p : After!ActivityPartition (
      name <- o.name,
      edge <- o.contents->select(c|c.oclIsKindOf(Before!Transition)),
      node <- o.contents->reject(c|c.oclIsKindOf(Before!ObjectFlowState))
    )
}

rule ActionState2OpaqueAction {
  from
    o : Before!ActionState
  to
    p : After!OpaqueAction (
      name <- o.name
    )
}

rule Initials {
  from
    o : Before!Pseudostate (
        o.kind = #inital
      )
  to
    p : After!InitialNode
}

rule Decisions {
  from
    o : Before!Pseudostate (
        o.kind = #junction
      )
  to
    p : After!DecisionNode
}

rule Forks {
  from
```

```
  o : Before!Pseudostate (
      o.kind = #fork
    )
 to
  p : After!ForkNode
}

rule Joins {
 from
  o : Before!Pseudostate (
      o.kind = #join
    )
 to
  p : After!MergeNode
}

rule Finals {
 from
  o : Before!FinalState
 to
  p : After!ActivityFinalNode
}

rule ObjectFlows {
 from
  o : Before!Transition (
   o.target.oclIsTypeOf(Before!ObjectFlowState)
  )
 to
  p : After!ObjectFlow (
   source <- o.source,
   target <- o.target.outgoing->first().target
  )
}

rule ControlFlows {
 from
  o : Before!Transition (
   not o.source.oclIsTypeOf(Before!ObjectFlowState) and
   not o.target.oclIsTypeOf(Before!ObjectFlowState)
  )
 to
  p : After!ControlFlow (
   guard <- o.guard,
   source <- o.source,
   target <- o.target
  )
```

```
}

rule Guards {
 from
  o : Before!Guard
 to
  p : After!OpaqueExpression (
   body <- o.expression.body
  )
}
```

Listing C.7: UML activity diagram migration in ATL

```
for (model in activities.Model.allInstances) {
 model.migrate(activities.Package)
 def ownedElement = model.unset(ownedElement)
 model.packagedElement = ownedElement
}

for (activity in activities.ActivityGraph.allInstances) {
 activity.migrate(activities.Activity)
 def top = activity.unset(top)
 activity.node = top.subvertex
 def transitions = activity.unset(transitions)
 activity.edge = transitions
 def partition = activity.unset(partition)
 activity.group = partition
}

for (partition in activities.ActivityGraph.allInstances) {
 def contents = partition.unset(contents)
 partition.edges = contents.findAll{it -> it instanceof activities.Transition}
 partition.nodes = contents.findAll{it -> it instanceof activities.StateVertex
   and not (it instanceof activities.ObjectFlowState)}
}

for (action in activities.ActionState.allInstances) {
 action.migrate(activities.OpaqueAction)
}

for (pseudostate in activities.Pseudeostate) {
 switch ( pseudostate.kind.toString() ) {
   case "pk_initial":
     pseudeostate.migrate(activities.InitialNode); break
  case "pk_junction"
   pseudeostate.migrate(activities.DecisionNode); break
  case "pk_fork"
```

```
   pseudeostate.migrate(activities.ForkNode); break
  case "pk_join"
   pseudeostate.migrate(activities.JoinNode); break
 }
}

for (finalstate in activities.FinalState.allInstances) {
 finalstate.migrate(activities.ActivityFinalNode)
}

for (transition in activities.ObjectFlow.allInstances.findAll{it -> it.target
   instanceof activities.ObjectFlowState}) {
 transition.target = transition.target.outgoing.first.target
}

for (transition in activities.Transition.allInstances) {
 transition.migrate(activities.ControlFlow)
}

for (guard in activities.Guard.allInstances) {
 transition.migrate(activities.OpaqueExpression)
 def expression = transition.unset(expression)
 transition.body = expression.body
}
```

Listing C.8: UML activity diagram migration in Groovy-for-COPE

```
migrate Model to Package {
 migrated.packagedElement := original.ownedElement.equivalent();
}

migrate ActivityGraph to Activity {
 migrated.node := original.top.subvertex.equivalent();
 migrated.edge := original.transitions.equivalent();
}

migrate Partition to ActivityPartition {
 migrated.edges := original.contents.collect(e : Transition | e.equivalent());
 migrated.nodes := original.contents.reject(ofs : ObjectFlowState | true).
   collect(n : StateVertex | n.equivalent());
}

migrate ActionState to OpaqueAction

migrate Pseudostate to InitialNode when: original.kind.toString() = 'pk_initial'
migrate Pseudostate to DecisionNode when: original.kind.toString() = '
   pk_junction'
```

```
migrate Pseudostate to ForkNode when: original.kind.toString() = 'pk_fork'
migrate Pseudostate to JoinNode when: original.kind.toString() = 'pk_join'

migrate FinalState to ActivityFinalNode

migrate Transition to ObjectFlow when: original.target.isTypeOf(ObjectFlowState)
     {
  migrated.source := original.source.equivalent();
  migrated.target := original.target.outgoing.first.target.equivalent();
}

migrate Transition to ControlFlow

migrate Guard to OpaqueExpression {
  migrated.body.add(original.expression.body);
}
```

Listing C.9: UML activity diagram migration in Flock

## C.3 GMF Examples

Two co-evolution examples were located in the Graphical Modeling Framework (GMF) project [Gronback 2009]. GMF allows the specification of a graphical concrete syntax for metamodel and the generation of graphical model editors from a number of graphical concrete syntax models. GMF was discussed in Section 2.3, and used to implement the graphical editor described in Appendix B.

GMF is implemented in a model-driven manner, and uses several metamodels to describe graphical concrete syntax and graphical model editors. During the development of GMF, two of its metamodels have evolved in a manner that has required models to be migrated. This section describes changes to the GMF Graph metamodel (used to describe the canvas of a graphical model editor) and the GMF Generator metamodel (used to describe the Java code generated for a graphical model editor).

### C.3.1 GMF Graph

The GMF Graph metamodel comprises approximately 60 classes. For clarity, only those classes that were affected by the changes made between versions 1.0 and 2.0 of GMF are shown in Figure C.4. The migration strategies were specified on the complete metamodel, and not only the extract shown here.

The GMF Graph metamodel (Figure C.4) describes the appearance of the generated graphical model editor. The metaclasses `Canvas`, `Figure`, `Node`, `DiagramLabel`, `Connection`, and `Compartment` are used to represent components of the graphical model editor to be generated. The evolution in the GMF Graph metamodel was driven by analysing the usage of the `Figure#referencingElements` reference, which relates `Figures` to the `DiagramElements` that use them. As described in the GMF

Graph documentation[2], the `referencingElements` reference increased the effort required to re-use figures, a common activity for users of GMF. Furthermore, `referencingElements` was used only by the GMF code generator to determine whether an accessor should be generated for nested `Figures`.

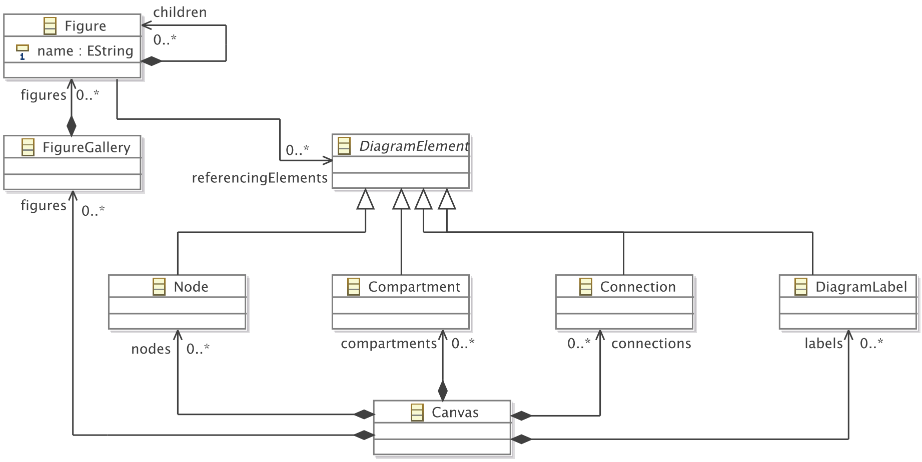
During the development of GMF 2.0, the Graph metamodel from GMF 1.0 was evolved – as shown in Figure 6.15(b) – to facilitate greater re-use of figures by introducing a proxy [Gamma *et al.* 1995] for `Figure`, termed `FigureDescriptor`. The original `referencingElements` reference was removed, and an extra metaclass, `ChildAccess`, was added to make more explicit the original purpose of `referencingElements` (accessing nested `Figures`).

Listings C.10, C.11 and C.12 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved creating proxy objects for the `FigureGallery#descriptors` and `FigureDescriptor#accessors` features, and moving values to those proxy objects.

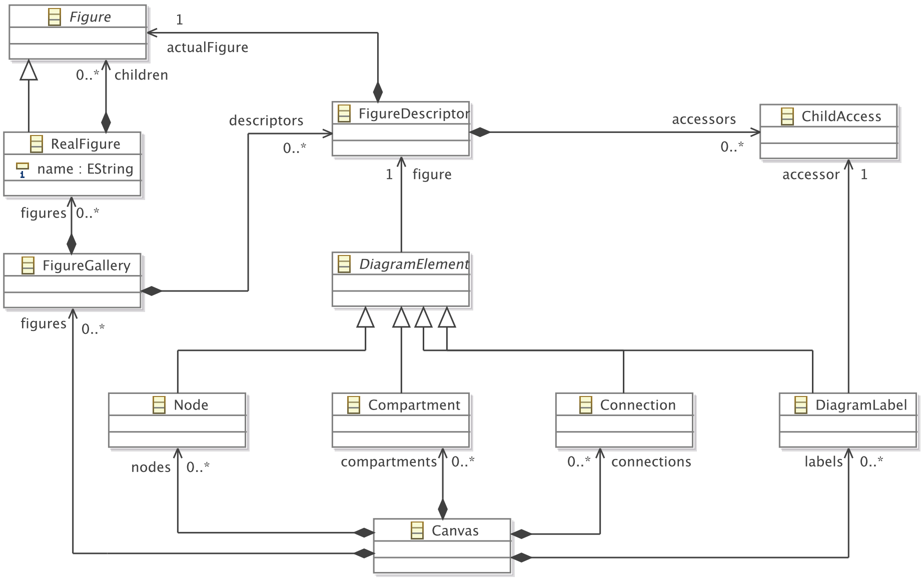
```
module Graph;

create Migrated : After from Original : Before;

rule Canvas2Canvas extends Identity2Identity {
  from
    o : Before!Canvas
  to
    m : After!Canvas (
      figures <- o.figures,
      nodes <- o.nodes,
      connections <- o.connections,
      compartments <- o.compartments,
      labels <- o.labels
    )
}
rule FigureGallery2FigureGallery extends Identity2Identity {
  from
    o : Before!FigureGallery
  to
    m : After!FigureGallery (
       implementationBundle <- o.implementationBundle
    )
}
abstract rule Identity2Identity {
  from
    o : Before!Identity
  to
    m : After!Identity (
       name <- o.name
```

---

[2] `http://wiki.eclipse.org/GMFGraph_Hints`



(a) Original metamodel.



(b) Evolved metamodel.

Figure C.4: The Graph metamodel in GMF 1.0 and GMF 2.0

```
   )
}
abstract rule DiagramElement2DiagramElement extends Identity2Identity {
 from
   o : Before!DiagramElement
 to
   m : After!DiagramElement (
     figure <- o.figure,
     facets <- o.facets
   )
}
rule Node2Node extends DiagramElement2DiagramElement {
 from
   o : Before!Node
 to
   m : After!Node (
     resizeConstraint <- o.resizeConstraint,
     affixedParentSide <- o.affixedParentSide
   )
}
rule Connection2Connection extends DiagramElement2DiagramElement {
 from
   o : Before!Connection
 to
   m : After!Connection
}
rule Compartment2Compartment extends DiagramElement2DiagramElement {
 from
   o : Before!Compartment
 to
   m : After!Compartment (
     collapsible <- o.collapsible,
     needsTitle <- o.needsTitle
   )
}
rule DiagramLabel2DiagramLabel extends Node2Node {
 from
   o : Before!DiagramLabel
 to
   m : After!DiagramLabel (
     elementIcon <- o.elementIcon
   )
}
abstract rule VisualFacet2VisualFacet {
 from
   o : Before!VisualFacet
 to
```

```
   m : After!VisualFacet
}
rule GeneralFacet2GeneralFacet extends VisualFacet2VisualFacet {
 from
   o : Before!GeneralFacet
 to
   m : After!GeneralFacet (
     identifier <- o.identifier,
     data <- o.data
   )
}
rule AlignmentFacet2AlignmentFacet extends VisualFacet2VisualFacet {
 from
   o : Before!AlignmentFacet
 to
   m : After!AlignmentFacet (
     alignment <- o.alignment
   )
}
rule GradientFacet2GradientFacet extends VisualFacet2VisualFacet {
 from
   o : Before!GradientFacet
 to
   m : After!GradientFacet (
     direction <- o.direction
   )
}
rule LabelOffsetFacet2LabelOffsetFacet extends VisualFacet2VisualFacet {
 from
   o : Before!LabelOffsetFacet
 to
   m : After!LabelOffsetFacet (
     x <- o.x,
     y <- o.y
   )
}
rule DefaultSizeFacet2DefaultSizeFacet extends VisualFacet2VisualFacet {
 from
   o : Before!DefaultSizeFacet
 to
   m : After!DefaultSizeFacet (
     defaultSize <- o.defaultSize
   )
}
abstract rule Figure2Figure extends Layoutable2Layoutable {
 from
   o : Before!Figure
```

```
 to
   m : After!Figure (
     foregroundColor <- o.foregroundColor,
     backgroundColor <- o.backgroundColor,
     maximumSize <- o.maximumSize,
     minimumSize <- o.minimumSize,
     preferredSize <- o.preferredSize,
     font <- o.font,
     insets <- o.insets,
     border <- o.border,
     location <- o.location,
     size <- o.size
   )
}
rule FigureRef2FigureRef extends Layoutable2Layoutable {
 from
   o : Before!FigureRef
 to
   m : After!FigureRef (
     figure <- o.figure
   )
}
abstract rule Shape2Shape extends Figure2Figure {
 from
   o : Before!Shape
 to
   m : After!Shape (
     outline <- o.outline,
     fill <- o.fill,
     lineWidth <- o.lineWidth,
     lineKind <- o.lineKind,
     xorFill <- o.xorFill,
     xorOutline <- o.xorOutline,
     resolvedChildren <- o.resolvedChildren
   )
}
rule Label2Label extends Figure2Figure {
 from
   o : Before!Label
 to
   m : After!Label (
     text <- o.text
   )
}
rule LabeledContainer2LabeledContainer extends Figure2Figure {
```

```
  from
    o : Before!LabeledContainer
  to
    m : After!LabeledContainer
}
rule Rectangle2Rectangle extends Shape2Shape {
  from
    o : Before!Rectangle
  to
    m : After!Rectangle
}
rule RoundedRectangle2RoundedRectangle extends Shape2Shape {
  from
    o : Before!RoundedRectangle
  to
    m : After!RoundedRectangle (
      cornerWidth <- o.cornerWidth,
      cornerHeight <- o.cornerHeight
    )
}
rule Ellipse2Ellipse extends Shape2Shape {
  from
    o : Before!Ellipse
  to
    m : After!Ellipse
}
rule Polyline2Polyline extends Shape2Shape {
  from
    o : Before!Polyline
  to
    m : After!Polyline (
      template <- o.template
    )
}
rule Polygon2Polygon extends Polyline2Polyline {
  from
    o : Before!Polygon
  to
    m : After!Polygon
}
rule ScalablePolygon2ScalablePolygon extends Polygon2Polygon {
  from
    o : Before!ScalablePolygon
  to
    m : After!ScalablePolygon
}
rule PolylineConnection2PolylineConnection extends Polyline2Polyline {
```

```
  from
    o : Before!PolylineConnection
  to
    m : After!PolylineConnection (
      sourceDecoration <- o.sourceDecoration,
      targetDecoration <- o.targetDecoration
    )
}
rule PolylineDecoration2PolylineDecoration extends Polyline2Polyline {
  from
    o : Before!PolylineDecoration
  to
    m : After!PolylineDecoration
}
rule PolygonDecoration2PolygonDecoration extends Polygon2Polygon {
  from
    o : Before!PolygonDecoration
  to
    m : After!PolygonDecoration
}
abstract rule CustomClass2CustomClass {
  from
    o : Before!CustomClass
  to
    m : After!CustomClass (
      qualifiedClassName <- o.qualifiedClassName,
      attributes <- o.attributes
    )
}
rule CustomAttribute2CustomAttribute {
  from
    o : Before!CustomAttribute
  to
    m : After!CustomAttribute (
      name <- o.name,
      value <- o.value,
      directAccess <- o.directAccess,
      multiStatementValue <- o.multiStatementValue
    )
}
rule FigureAccessor2FigureAccessor {
  from
    o : Before!FigureAccessor
  to
    m : After!FigureAccessor (
      accessor <- o.accessor,
```

```
      typedFigure <- o.typedFigure
    )
}
rule CustomFigure2CustomFigure extends Figure2Figure {
  from
    o : Before!CustomFigure
  to
    m : After!CustomFigure (
      customChildren <- o.customChildren
    )
}
rule CustomDecoration2CustomDecoration extends CustomFigure2CustomFigure {
  from
    o : Before!CustomDecoration
  to
    m : After!CustomDecoration
}
rule CustomConnection2CustomConnection extends CustomFigure2CustomFigure {
  from
    o : Before!CustomConnection
  to
    m : After!CustomConnection
}
abstract rule Color2Color {
  from
    o : Before!Color
  to
    m : After!Color
}
rule RGBColor2RGBColor extends Color2Color {
  from
    o : Before!RGBColor
  to
    m : After!RGBColor (
      red <- o.red,
      green <- o.green,
      blue <- o.blue
    )
}
rule ConstantColor2ConstantColor extends Color2Color {
  from
    o : Before!ConstantColor
  to
    m : After!ConstantColor (
      value <- o.value
    )
}
```

```
abstract rule Font2Font {
 from
   o : Before!Font
 to
   m : After!Font
}
rule BasicFont2BasicFont extends Font2Font {
 from
   o : Before!BasicFont
 to
   m : After!BasicFont (
     faceName <- o.faceName,
     height <- o.height,
     style <- o.style
   )
}
rule Point2Point {
 from
   o : Before!Point
 to
   m : After!Point (
     x <- o.x,
     y <- o.y
   )
}
rule Dimension2Dimension {
 from
   o : Before!Dimension
 to
   m : After!Dimension (
     dx <- o.dx,
     dy <- o.dy
   )
}
rule Insets2Insets {
 from
   o : Before!Insets
 to
   m : After!Insets (
     top <- o.top,
     left <- o.left,
     bottom <- o.bottom,
     right <- o.right
   )
}
abstract rule Border2Border {
```

```
 from
   o : Before!Border
 to
   m : After!Border
}
rule LineBorder2LineBorder extends Border2Border {
 from
   o : Before!LineBorder
 to
   m : After!LineBorder (
     color <- o.color,
     width <- o.width
   )
}
rule MarginBorder2MarginBorder extends Border2Border {
 from
   o : Before!MarginBorder
 to
   m : After!MarginBorder (
     insets <- o.insets
   )
}
rule CompoundBorder2CompoundBorder extends Border2Border {
 from
   o : Before!CompoundBorder
 to
   m : After!CompoundBorder (
     outer <- o.outer,
     inner <- o.inner
   )
}
rule CustomBorder2CustomBorder extends Border2Border {
 from
   o : Before!CustomBorder
 to
   m : After!CustomBorder
}
abstract rule LayoutData2LayoutData {
 from
   o : Before!LayoutData
 to
   m : After!LayoutData (
     owner <- o.owner
   )
}
rule CustomLayoutData2CustomLayoutData extends LayoutData2LayoutData {
 from
```

```
  o : Before!CustomLayoutData
 to
  m : After!CustomLayoutData
}
rule GridLayoutData2GridLayoutData extends LayoutData2LayoutData {
 from
  o : Before!GridLayoutData
 to
  m : After!GridLayoutData (
    grabExcessHorizontalSpace <- o.grabExcessHorizontalSpace,
    grabExcessVerticalSpace <- o.grabExcessVerticalSpace,
    verticalAlignment <- o.verticalAlignment,
    horizontalAlignment <- o.horizontalAlignment,
    verticalSpan <- o.verticalSpan,
    horizontalSpan <- o.horizontalSpan,
    horizontalIndent <- o.horizontalIndent,
    sizeHint <- o.sizeHint
  )
}
rule BorderLayoutData2BorderLayoutData extends LayoutData2LayoutData {
 from
  o : Before!BorderLayoutData
 to
  m : After!BorderLayoutData (
    alignment <- o.alignment,
    vertical <- o.vertical
  )
}
abstract rule Layoutable2Layoutable {
 from
  o : Before!Layoutable
 to
  m : After!Layoutable (
    layoutData <- o.layoutData,
    layout <- o.layout
  )
}
abstract rule Layout2Layout {
 from
  o : Before!Layout
 to
  m : After!Layout
}
rule CustomLayout2CustomLayout extends Layout2Layout {
 from
  o : Before!CustomLayout
```

```
 to
   m : After!CustomLayout
}
rule GridLayout2GridLayout extends Layout2Layout {
 from
   o : Before!GridLayout
 to
   m : After!GridLayout (
     numColumns <- o.numColumns,
     equalWidth <- o.equalWidth,
     margins <- o.margins,
     spacing <- o.spacing
   )
}
rule BorderLayout2BorderLayout extends Layout2Layout {
 from
   o : Before!BorderLayout
 to
   m : After!BorderLayout (
     spacing <- o.spacing
   )
}
rule FlowLayout2FlowLayout extends Layout2Layout {
 from
   o : Before!FlowLayout
 to
   m : After!FlowLayout (
     vertical <- o.vertical,
     matchMinorSize <- o.matchMinorSize,
     forceSingleLine <- o.forceSingleLine,
     majorAlignment <- o.majorAlignment,
     minorAlignment <- o.minorAlignment,
     majorSpacing <- o.majorSpacing,
     minorSpacing <- o.minorSpacing
   )
}
rule XYLayout2XYLayout extends Layout2Layout {
 from
   o : Before!XYLayout
 to
   m : After!XYLayout
}
rule XYLayoutData2XYLayoutData extends LayoutData2LayoutData {
 from
   o : Before!XYLayoutData
 to
```

```
    m : After!XYLayoutData (
      topLeft <- o.topLeft,
      size <- o.size
    )
}
rule StackLayout2StackLayout extends Layout2Layout {
  from
    o : Before!StackLayout
  to
    m : After!StackLayout
}
```

Listing C.10: GMF Graph migration in ATL

```
for (gallery in graph.FigureGallery.allInstances) {
  while(not gallery.figures.isEmpty()) {
    def figure = gallery.figures.first()
    def descriptor = graph.FigureDescriptor.newInstance()

    descriptor.name = figure.name
    descriptor.actualFigure = figure

    figure.set(descriptor, descriptor)

    figure.children.findAll{ it -> it instanceof graph.Label}.each do |it|
      def accessor = graph.ChildAccess.newInstance()

      accessor.figure = it
      descriptor.accessors.add(accessor)

      it.set(accessor, accessor)
    end

    return descriptor;
  }
}

for (diagramElement in graph.DiagramElement.allInstances()) {
    diagramElement.figure.unset(descriptor)
   diagramElement.figure = descriptor
}

for (diagramLabel in graph.DiagramLabel.allInstances()) {
    diagramElement.figure.unset(accessor)
   diagramElement.accessor = accessor
```

```
}
```

Listing C.11: GMF Graph migration in Groovy-for-COPE

```
migrate FigureGallery {
  while (not migrated.figures.isEmpty()) {
    migrated.descriptors.add(migrated.figures.first.createDescriptor());
  }
}

migrate Compartment {
  migrated.figure := original.figure.equivalent().~descriptor;
}

migrate Connection {
  migrated.figure := original.figure.equivalent().~descriptor;
}

migrate DiagramLabel {
  migrated.figure := original.figure.equivalent().~descriptor;
  migrated.accessor := original.figure.equivalent().~accessor;
}

migrate Node {
  migrated.figure := original.figure.equivalent().~descriptor;
}

operation Migrated!Figure createDescriptor() : Migrated!FigureDescriptor {
  var descriptor := new Migrated!FigureDescriptor;

  descriptor.name := self.name;
  descriptor.actualFigure := self;

  self.~descriptor := descriptor;

  self.children.forAll(l : Migrated!Label | l.addAccessor(descriptor));

  return descriptor;
}

operation Migrated!Label addAccessor(descriptor : Migrated!FigureDescriptor) {
  var accessor := new Migrated!ChildAccess;

  accessor.figure := self;
  self.~descriptor := descriptor;
  self.~accessor := accessor;
  descriptor.accessors.add(accessor);
```

```
}
```

Listing C.12: GMF Graph migration in Flock

### C.3.2 GMF Generator

During the development of GMF v2.2, the Generator metamodel evolved to make explicit the use of `ContextMenus` and `Parsers`. In previous versions of GMF, `ContextMenus` and `Parsers` were not customisable via the Generator metamodel. Instead, the GMF runtime created menus and parsers automatically at runtime. The GMF generator metamodel is too large to show here, as it comprises approximately 150 classes and the changes made between versions 2.1 and 2.2 of GMF directly affected 23 classes.

Listings C.13, C.13 and C.13 show the model migration strategy in ATL, COPE and Flock respectively. Migration involved populating `ContextMenus` from existing diagram elements, and creating `Parsers` for built-in and user-defined languages.

```
module GenModel2009;

create Migrated : After from Original : Before;

rule GenEditorGenerator2GenEditorGenerator {
  from
    o : Before!GenEditorGenerator
  to
    m : After!GenEditorGenerator (
      audits <- o.audits,
      metrics <- o.metrics,
      diagram <- o.diagram,
      plugin <- o.plugin,
      editor <- o.editor,
      navigator <- o.navigator,
      diagramUpdater <- o.diagramUpdater,
      propertySheet <- o.propertySheet,
      application <- o.application,
      domainGenModel <- o.domainGenModel,
      packageNamePrefix <- o.packageNamePrefix,
      modelID <- o.modelID,
      sameFileForDiagramAndModel <- o.sameFileForDiagramAndModel,
      diagramFileExtension <- o.diagramFileExtension,
      domainFileExtension <- o.domainFileExtension,
      dynamicTemplates <- o.dynamicTemplates,
      templateDirectory <- o.templateDirectory,
      copyrightText <- o.copyrightText,
      expressionProviders <- o.expressionProviders,
```

```
    modelAccess <- o.modelAccess
  )
}
rule GenDiagram2GenDiagram extends GenContainerBase2GenContainerBase {
 from
  o : Before!GenDiagram
 to
  m : After!GenDiagram (
    domainDiagramElement <- o.domainDiagramElement,
    childNodes <- o.childNodes,
    topLevelNodes <- o.topLevelNodes,
    links <- o.links,
    compartments <- o.compartments,
    palette <- o.palette,
    synchronized <- o.synchronized,
    preferences <- o.preferences,
    preferencePages <- o.preferencePages
  )
}
rule GenEditorView2GenEditorView {
 from
  o : Before!GenEditorView
 to
  m : After!GenEditorView (
    packageName <- o.packageName,
    actionBarContributorClassName <- o.actionBarContributorClassName,
    className <- o.className,
    iconPath <- o.iconPath,
    iD <- o.iD,
    eclipseEditor <- o.eclipseEditor,
    contextID <- o.contextID
  )
}
abstract rule GenPreferencePage2GenPreferencePage {
 from
  o : Before!GenPreferencePage
 to
  m : After!GenPreferencePage (
    iD <- o.iD,
    name <- o.name,
    children <- o.children
  )
}
rule GenCustomPreferencePage2GenCustomPreferencePage extends
    GenPreferencePage2GenPreferencePage {
```

```
  from
    o : Before!GenCustomPreferencePage
  to
    m : After!GenCustomPreferencePage (
      qualifiedClassName <- o.qualifiedClassName
    )
}
rule GenStandardPreferencePage2GenStandardPreferencePage extends
    GenPreferencePage2GenPreferencePage {
  from
    o : Before!GenStandardPreferencePage
  to
    m : After!GenStandardPreferencePage (
      kind <- o.kind
    )
}
rule GenDiagramPreferences2GenDiagramPreferences {
  from
    o : Before!GenDiagramPreferences
  to
    m : After!GenDiagramPreferences (
      lineStyle <- o.lineStyle,
      defaultFont <- o.defaultFont,
      fontColor <- o.fontColor,
      fillColor <- o.fillColor,
      lineColor <- o.lineColor,
      noteFillColor <- o.noteFillColor,
      noteLineColor <- o.noteLineColor,
      showConnectionHandles <- o.showConnectionHandles,
      showPopupBars <- o.showPopupBars,
      promptOnDelFromModel <- o.promptOnDelFromModel,
      promptOnDelFromDiagram <- o.promptOnDelFromDiagram,
      enableAnimatedLayout <- o.enableAnimatedLayout,
      enableAnimatedZoom <- o.enableAnimatedZoom,
      enableAntiAlias <- o.enableAntiAlias,
      showGrid <- o.showGrid,
      showRulers <- o.showRulers,
      snapToGrid <- o.snapToGrid,
      snapToGeometry <- o.snapToGeometry,
      gridInFront <- o.gridInFront,
      rulerUnits <- o.rulerUnits,
      gridSpacing <- o.gridSpacing,
      gridLineColor <- o.gridLineColor,
      gridLineStyle <- o.gridLineStyle
    )
```

```
}
abstract rule GenFont2GenFont {
  from
    o : Before!GenFont
  to
    m : After!GenFont
}
rule GenStandardFont2GenStandardFont extends GenFont2GenFont {
  from
    o : Before!GenStandardFont
  to
    m : After!GenStandardFont (
      name <- o.name
    )
}
rule GenCustomFont2GenCustomFont extends GenFont2GenFont {
  from
    o : Before!GenCustomFont
  to
    m : After!GenCustomFont (
      name <- o.name,
      height <- o.height,
      style <- o.style
    )
}
abstract rule GenColor2GenColor {
  from
    o : Before!GenColor
  to
    m : After!GenColor
}
rule GenRGBColor2GenRGBColor extends GenColor2GenColor {
  from
    o : Before!GenRGBColor
  to
    m : After!GenRGBColor (
      red <- o.red,
      green <- o.green,
      blue <- o.blue
    )
}
rule GenConstantColor2GenConstantColor extends GenColor2GenColor {
  from
    o : Before!GenConstantColor
  to
    m : After!GenConstantColor (
      name <- o.name
```

```
   )
}
rule GenDiagramUpdater2GenDiagramUpdater {
 from
   o : Before!GenDiagramUpdater
 to
   m : After!GenDiagramUpdater (
     diagramUpdaterClassName <- o.diagramUpdaterClassName,
     nodeDescriptorClassName <- o.nodeDescriptorClassName,
     linkDescriptorClassName <- o.linkDescriptorClassName,
     updateCommandClassName <- o.updateCommandClassName,
     updateCommandID <- o.updateCommandID
   )
}
rule GenPlugin2GenPlugin {
 from
   o : Before!GenPlugin
 to
   m : After!GenPlugin (
     iD <- o.iD,
     name <- o.name,
     provider <- o.provider,
     version <- o.version,
     printingEnabled <- o.printingEnabled,
     requiredPlugins <- o.requiredPlugins,
     activatorClassName <- o.activatorClassName
   )
}
rule DynamicModelAccess2DynamicModelAccess {
 from
   o : Before!DynamicModelAccess
 to
   m : After!DynamicModelAccess (
     packageName <- o.packageName,
     className <- o.className
   )
}
abstract rule GenCommonBase2GenCommonBase {
 from
   o : Before!GenCommonBase
 to
   m : After!GenCommonBase (
     diagramRunTimeClass <- o.diagramRunTimeClass,
     visualID <- o.visualID,
     elementType <- o.elementType,
     editPartClassName <- o.editPartClassName,
```

```
      itemSemanticEditPolicyClassName <- o.itemSemanticEditPolicyClassName,
      notationViewFactoryClassName <- o.notationViewFactoryClassName,
      viewmap <- o.viewmap,
      styles <- o.styles,
      behaviour <- o.behaviour
    )
}
abstract rule Behaviour2Behaviour {
  from
    o : Before!Behaviour
  to
    m : After!Behaviour
}
rule CustomBehaviour2CustomBehaviour extends Behaviour2Behaviour {
  from
    o : Before!CustomBehaviour
  to
    m : After!CustomBehaviour (
      key <- o.key,
      editPolicyQualifiedClassName <- o.editPolicyQualifiedClassName
    )
}
rule SharedBehaviour2SharedBehaviour extends Behaviour2Behaviour {
  from
    o : Before!SharedBehaviour
  to
    m : After!SharedBehaviour (
      delegate <- o.delegate
    )
}
rule OpenDiagramBehaviour2OpenDiagramBehaviour extends Behaviour2Behaviour {
  from
    o : Before!OpenDiagramBehaviour
  to
    m : After!OpenDiagramBehaviour (
      editPolicyClassName <- o.editPolicyClassName,
      diagramKind <- o.diagramKind,
      editorID <- o.editorID,
      openAsEclipseEditor <- o.openAsEclipseEditor
    )
}
abstract rule GenContainerBase2GenContainerBase extends
    GenCommonBase2GenCommonBase {
  from
    o : Before!GenContainerBase
  to
```

```
    m : After!GenContainerBase (
      canonicalEditPolicyClassName <- o.canonicalEditPolicyClassName
    )
}
abstract rule GenChildContainer2GenChildContainer extends GenContainerBase2GenContainerBase {
  from
    o : Before!GenChildContainer
  to
    m : After!GenChildContainer (
      childNodes <- o.childNodes
    )
}
abstract rule GenNode2GenNode extends GenChildContainer2GenChildContainer {
  from
    o : Before!GenNode
  to
    m : After!GenNode (
      modelFacet <- o.modelFacet,
      labels <- o.labels,
      compartments <- o.compartments,
      primaryDragEditPolicyQualifiedClassName <- o.primaryDragEditPolicyQualifiedClassName,
      graphicalNodeEditPolicyClassName <- o.graphicalNodeEditPolicyClassName,
      createCommandClassName <- o.createCommandClassName
    )
}
rule GenTopLevelNode2GenTopLevelNode extends GenNode2GenNode {
  from
    o : Before!GenTopLevelNode
  to
    m : After!GenTopLevelNode
}
rule GenChildNode2GenChildNode extends GenNode2GenNode {
  from
    o : Before!GenChildNode
  to
    m : After!GenChildNode
}
rule GenChildSideAffixedNode2GenChildSideAffixedNode extends GenChildNode2GenChildNode {
  from
    o : Before!GenChildSideAffixedNode
  to
    m : After!GenChildSideAffixedNode (
      preferredSideName <- o.preferredSideName
```

```
    )
}
rule GenChildLabelNode2GenChildLabelNode extends GenChildNode2GenChildNode {
  from
    o : Before!GenChildLabelNode
  to
    m : After!GenChildLabelNode (
      labelReadOnly <- o.labelReadOnly,
      labelElementIcon <- o.labelElementIcon,
      labelModelFacet <- o.labelModelFacet
    )
}
rule GenCompartment2GenCompartment extends GenChildContainer2GenChildContainer {
  from
    o : Before!GenCompartment
  to
    m : After!GenCompartment (
      title <- o.title,
      canCollapse <- o.canCollapse,
      hideIfEmpty <- o.hideIfEmpty,
      needsTitle <- o.needsTitle,
      node <- o.node,
      listLayout <- o.listLayout
    )
}
rule GenLink2GenLink extends GenCommonBase2GenCommonBase {
  from
    o : Before!GenLink
  to
    m : After!GenLink (
      modelFacet <- o.modelFacet,
      labels <- o.labels,
      outgoingCreationAllowed <- o.outgoingCreationAllowed,
      incomingCreationAllowed <- o.incomingCreationAllowed,
      viewDirectionAlignedWithModel <- o.viewDirectionAlignedWithModel,
      creationConstraints <- o.creationConstraints,
      createCommandClassName <- o.createCommandClassName,
      reorientCommandClassName <- o.reorientCommandClassName,
      treeBranch <- o.treeBranch
    )
}
abstract rule GenLabel2GenLabel extends GenCommonBase2GenCommonBase {
  from
    o : Before!GenLabel
  to
    m : After!GenLabel (
```

```
      readOnly <- o.readOnly,
      elementIcon <- o.elementIcon,
      modelFacet <- o.modelFacet
    )
}
rule GenNodeLabel2GenNodeLabel extends GenLabel2GenLabel {
  from
    o : Before!GenNodeLabel
  to
    m : After!GenNodeLabel
}
rule GenExternalNodeLabel2GenExternalNodeLabel extends GenNodeLabel2GenNodeLabel
     {
  from
    o : Before!GenExternalNodeLabel
  to
    m : After!GenExternalNodeLabel
}
rule GenLinkLabel2GenLinkLabel extends GenLabel2GenLabel {
  from
    o : Before!GenLinkLabel
  to
    m : After!GenLinkLabel (
      link <- o.link,
      alignment <- o.alignment
    )
}
abstract rule ElementType2ElementType {
  from
    o : Before!ElementType
  to
    m : After!ElementType (
      diagramElement <- o.diagramElement,
      uniqueIdentifier <- o.uniqueIdentifier,
      displayName <- o.displayName,
      definedExternally <- o.definedExternally
    )
}
rule MetamodelType2MetamodelType extends ElementType2ElementType {
  from
    o : Before!MetamodelType
  to
    m : After!MetamodelType (
      editHelperClassName <- o.editHelperClassName
    )
}
```

```
rule SpecializationType2SpecializationType extends ElementType2ElementType {
 from
  o : Before!SpecializationType
 to
  m : After!SpecializationType (
   metamodelType <- o.metamodelType,
   editHelperAdviceClassName <- o.editHelperAdviceClassName
  )
}
rule NotationType2NotationType extends ElementType2ElementType {
 from
  o : Before!NotationType
 to
  m : After!NotationType
}
abstract rule ModelFacet2ModelFacet {
 from
  o : Before!ModelFacet
 to
  m : After!ModelFacet
}
abstract rule LinkModelFacet2LinkModelFacet extends ModelFacet2ModelFacet {
 from
  o : Before!LinkModelFacet
 to
  m : After!LinkModelFacet
}
abstract rule LabelModelFacet2LabelModelFacet extends ModelFacet2ModelFacet {
 from
  o : Before!LabelModelFacet
 to
  m : After!LabelModelFacet
}
rule TypeModelFacet2TypeModelFacet extends ModelFacet2ModelFacet {
 from
  o : Before!TypeModelFacet
 to
  m : After!TypeModelFacet (
   metaClass <- o.metaClass,
   containmentMetaFeature <- o.containmentMetaFeature,
   childMetaFeature <- o.childMetaFeature,
   modelElementSelector <- o.modelElementSelector,
   modelElementInitializer <- o.modelElementInitializer
  )
}
rule TypeLinkModelFacet2TypeLinkModelFacet extends TypeModelFacet2TypeModelFacet
     {
```

```
 from
   o : Before!TypeLinkModelFacet
 to
   m : After!TypeLinkModelFacet (
     sourceMetaFeature <- o.sourceMetaFeature,
     targetMetaFeature <- o.targetMetaFeature
   )
}
rule FeatureLinkModelFacet2FeatureLinkModelFacet extends
    LinkModelFacet2LinkModelFacet {
 from
   o : Before!FeatureLinkModelFacet
 to
   m : After!FeatureLinkModelFacet (
     metaFeature <- o.metaFeature
   )
}
rule FeatureLabelModelFacet2FeatureLabelModelFacet extends
    LabelModelFacet2LabelModelFacet {
 from
   o : Before!FeatureLabelModelFacet
 to
   m : After!FeatureLabelModelFacet (
     metaFeatures <- o.metaFeatures,
     viewPattern <- o.viewPattern,
     editorPattern <- o.editorPattern,
     editPattern <- o.editPattern,
     viewMethod <- o.viewMethod,
     editMethod <- o.editMethod
   )
}
rule DesignLabelModelFacet2DesignLabelModelFacet extends
    LabelModelFacet2LabelModelFacet {
 from
   o : Before!DesignLabelModelFacet
 to
   m : After!DesignLabelModelFacet
}
abstract rule Attributes2Attributes {
 from
   o : Before!Attributes
 to
   m : After!Attributes
}
rule ColorAttributes2ColorAttributes extends Attributes2Attributes {
 from
```

```
    o : Before!ColorAttributes
  to
    m : After!ColorAttributes (
      foregroundColor <- o.foregroundColor,
      backgroundColor <- o.backgroundColor
    )
}
rule StyleAttributes2StyleAttributes extends Attributes2Attributes {
  from
    o : Before!StyleAttributes
  to
    m : After!StyleAttributes (
      fixedFont <- o.fixedFont,
      fixedForeground <- o.fixedForeground,
      fixedBackground <- o.fixedBackground
    )
}
rule ResizeConstraints2ResizeConstraints extends Attributes2Attributes {
  from
    o : Before!ResizeConstraints
  to
    m : After!ResizeConstraints (
      resizeHandles <- o.resizeHandles,
      nonResizeHandles <- o.nonResizeHandles
    )
}
rule DefaultSizeAttributes2DefaultSizeAttributes extends Attributes2Attributes {
  from
    o : Before!DefaultSizeAttributes
  to
    m : After!DefaultSizeAttributes (
      width <- o.width,
      height <- o.height
    )
}
rule LabelOffsetAttributes2LabelOffsetAttributes extends Attributes2Attributes {
  from
    o : Before!LabelOffsetAttributes
  to
    m : After!LabelOffsetAttributes (
      x <- o.x,
      y <- o.y
    )
}
abstract rule Viewmap2Viewmap {
  from
```

```
    o : Before!Viewmap
  to
    m : After!Viewmap (
      attributes <- o.attributes,
      requiredPluginIDs <- o.requiredPluginIDs,
      layoutType <- o.layoutType
    )
}
rule FigureViewmap2FigureViewmap extends Viewmap2Viewmap {
  from
    o : Before!FigureViewmap
  to
    m : After!FigureViewmap (
      figureQualifiedClassName <- o.figureQualifiedClassName
    )
}
rule SnippetViewmap2SnippetViewmap extends Viewmap2Viewmap {
  from
    o : Before!SnippetViewmap
  to
    m : After!SnippetViewmap (
      body <- o.body
    )
}
rule InnerClassViewmap2InnerClassViewmap extends Viewmap2Viewmap {
  from
    o : Before!InnerClassViewmap
  to
    m : After!InnerClassViewmap (
      className <- o.className,
      classBody <- o.classBody
    )
}
rule ParentAssignedViewmap2ParentAssignedViewmap extends Viewmap2Viewmap {
  from
    o : Before!ParentAssignedViewmap
  to
    m : After!ParentAssignedViewmap (
      getterName <- o.getterName,
      setterName <- o.setterName,
      figureQualifiedClassName <- o.figureQualifiedClassName
    )
}
rule ValueExpression2ValueExpression {
  from
    o : Before!ValueExpression
```

```
 to
   m : After!ValueExpression (
     body <- o.body
   )
}
rule GenConstraint2GenConstraint extends ValueExpression2ValueExpression {
 from
   o : Before!GenConstraint
 to
   m : After!GenConstraint
}
rule Palette2Palette {
 from
   o : Before!Palette
 to
   m : After!Palette (
     flyout <- o.flyout,
     groups <- o.groups,
     packageName <- o.packageName,
     factoryClassName <- o.factoryClassName
   )
}
abstract rule EntryBase2EntryBase {
 from
   o : Before!EntryBase
 to
   m : After!EntryBase (
     title <- o.title,
     description <- o.description,
     largeIconPath <- o.largeIconPath,
     smallIconPath <- o.smallIconPath,
     createMethodName <- o.createMethodName
   )
}
abstract rule AbstractToolEntry2AbstractToolEntry extends EntryBase2EntryBase {
 from
   o : Before!AbstractToolEntry
 to
   m : After!AbstractToolEntry (
     default <- o.default,
     qualifiedToolName <- o.qualifiedToolName,
     properties <- o.properties
   )
}
rule ToolEntry2ToolEntry extends AbstractToolEntry2AbstractToolEntry {
 from
```

```
    o : Before!ToolEntry
  to
    m : After!ToolEntry (
      genNodes <- o.genNodes,
      genLinks <- o.genLinks
    )
}
rule StandardEntry2StandardEntry extends AbstractToolEntry2AbstractToolEntry {
  from
    o : Before!StandardEntry
  to
    m : After!StandardEntry (
      kind <- o.kind
    )
}
abstract rule ToolGroupItem2ToolGroupItem {
  from
    o : Before!ToolGroupItem
  to
    m : After!ToolGroupItem
}
rule Separator2Separator extends ToolGroupItem2ToolGroupItem {
  from
    o : Before!Separator
  to
    m : After!Separator
}
rule ToolGroup2ToolGroup extends EntryBase2EntryBase {
  from
    o : Before!ToolGroup
  to
    m : After!ToolGroup (
      palette <- o.palette,
      stack <- o.stack,
      collapse <- o.collapse,
      entries <- o.entries
    )
}
abstract rule GenElementInitializer2GenElementInitializer {
  from
    o : Before!GenElementInitializer
  to
    m : After!GenElementInitializer
}
rule GenFeatureSeqInitializer2GenFeatureSeqInitializer extends
     GenElementInitializer2GenElementInitializer {
  from
```

```
    o : Before!GenFeatureSeqInitializer
  to
    m : After!GenFeatureSeqInitializer (
      initializers <- o.initializers,
      elementClass <- o.elementClass
    )
}
rule GenFeatureValueSpec2GenFeatureValueSpec extends
    GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenFeatureValueSpec
  to
    m : After!GenFeatureValueSpec (
      value <- o.value
    )
}
rule GenReferenceNewElementSpec2GenReferenceNewElementSpec extends
    GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenReferenceNewElementSpec
  to
    m : After!GenReferenceNewElementSpec (
      newElementInitializers <- o.newElementInitializers
    )
}
abstract rule GenFeatureInitializer2GenFeatureInitializer {
  from
    o : Before!GenFeatureInitializer
  to
    m : After!GenFeatureInitializer (
      feature <- o.feature
    )
}
rule GenLinkConstraints2GenLinkConstraints {
  from
    o : Before!GenLinkConstraints
  to
    m : After!GenLinkConstraints (
      link <- o.link,
      sourceEnd <- o.sourceEnd,
      targetEnd <- o.targetEnd
    )
}
rule GenAuditRoot2GenAuditRoot {
  from
    o : Before!GenAuditRoot
  to
```

```
  m : After!GenAuditRoot (
    categories <- o.categories,
    rules <- o.rules,
    clientContexts <- o.clientContexts
  )
}
rule GenAuditContainer2GenAuditContainer {
 from
  o : Before!GenAuditContainer
 to
  m : After!GenAuditContainer (
    id <- o.id,
    name <- o.name,
    description <- o.description,
    path <- o.path,
    audits <- o.audits
  )
}
abstract rule GenRuleBase2GenRuleBase {
 from
  o : Before!GenRuleBase
 to
  m : After!GenRuleBase (
    name <- o.name,
    description <- o.description
  )
}
rule GenAuditRule2GenAuditRule extends GenRuleBase2GenRuleBase {
 from
  o : Before!GenAuditRule
 to
  m : After!GenAuditRule (
    id <- o.id,
    rule <- o.rule,
    target <- o.target,
    message <- o.message,
    severity <- o.severity,
    useInLiveMode <- o.useInLiveMode,
    category <- o.category
  )
}
abstract rule GenRuleTarget2GenRuleTarget {
 from
  o : Before!GenRuleTarget
 to
  m : After!GenRuleTarget
```

```
}
rule GenDomainElementTarget2GenDomainElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDomainElementTarget
  to
    m : After!GenDomainElementTarget (
      element <- o.element
    )
}
rule GenDiagramElementTarget2GenDiagramElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDiagramElementTarget
  to
    m : After!GenDiagramElementTarget (
      element <- o.element
    )
}
rule GenDomainAttributeTarget2GenDomainAttributeTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenDomainAttributeTarget
  to
    m : After!GenDomainAttributeTarget (
      attribute <- o.attribute,
      nullAsError <- o.nullAsError
    )
}
rule GenNotationElementTarget2GenNotationElementTarget extends
    GenAuditable2GenAuditable {
  from
    o : Before!GenNotationElementTarget
  to
    m : After!GenNotationElementTarget (
      element <- o.element
    )
}
rule GenMetricContainer2GenMetricContainer {
  from
    o : Before!GenMetricContainer
  to
    m : After!GenMetricContainer (
      metrics <- o.metrics
    )
}
rule GenMetricRule2GenMetricRule extends GenRuleBase2GenRuleBase {
```

```
  from
    o : Before!GenMetricRule
  to
    m : After!GenMetricRule (
      key <- o.key,
      rule <- o.rule,
      target <- o.target,
      lowLimit <- o.lowLimit,
      highLimit <- o.highLimit,
      container <- o.container
    )
}
rule GenAuditedMetricTarget2GenAuditedMetricTarget extends
     GenAuditable2GenAuditable {
  from
    o : Before!GenAuditedMetricTarget
  to
    m : After!GenAuditedMetricTarget (
      metric <- o.metric,
      metricValueContext <- o.metricValueContext
    )
}
abstract rule GenAuditable2GenAuditable extends GenRuleTarget2GenRuleTarget {
  from
    o : Before!GenAuditable
  to
    m : After!GenAuditable (
      contextSelector <- o.contextSelector
    )
}
rule GenAuditContext2GenAuditContext {
  from
    o : Before!GenAuditContext
  to
    m : After!GenAuditContext (
      root <- o.root,
      id <- o.id,
      className <- o.className,
      ruleTargets <- o.ruleTargets
    )
}
abstract rule GenMeasurable2GenMeasurable extends GenRuleTarget2GenRuleTarget {
  from
    o : Before!GenMeasurable
  to
    m : After!GenMeasurable
```

```
}
rule GenExpressionProviderContainer2GenExpressionProviderContainer {
  from
    o : Before!GenExpressionProviderContainer
  to
    m : After!GenExpressionProviderContainer (
      expressionsPackageName <- o.expressionsPackageName,
      abstractExpressionClassName <- o.abstractExpressionClassName,
      providers <- o.providers
    )
}
abstract rule GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenExpressionProviderBase
  to
    m : After!GenExpressionProviderBase (
      expressions <- o.expressions
    )
}
rule GenJavaExpressionProvider2GenJavaExpressionProvider extends
    GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenJavaExpressionProvider
  to
    m : After!GenJavaExpressionProvider (
      throwException <- o.throwException,
      injectExpressionBody <- o.injectExpressionBody
    )
}
rule GenExpressionInterpreter2GenExpressionInterpreter extends
    GenExpressionProviderBase2GenExpressionProviderBase {
  from
    o : Before!GenExpressionInterpreter
  to
    m : After!GenExpressionInterpreter (
      language <- o.language,
      className <- o.className
    )
}
abstract rule GenDomainModelNavigator2GenDomainModelNavigator {
  from
    o : Before!GenDomainModelNavigator
  to
    m : After!GenDomainModelNavigator (
      generateDomainModelNavigator <- o.generateDomainModelNavigator,
      domainContentExtensionID <- o.domainContentExtensionID,
```

```
    domainContentExtensionName <- o.domainContentExtensionName,
    domainContentExtensionPriority <- o.domainContentExtensionPriority,
    domainContentProviderClassName <- o.domainContentProviderClassName,
    domainLabelProviderClassName <- o.domainLabelProviderClassName,
    domainModelElementTesterClassName <- o.domainModelElementTesterClassName,
    domainNavigatorItemClassName <- o.domainNavigatorItemClassName
   )
}
rule GenNavigator2GenNavigator extends
    GenDomainModelNavigator2GenDomainModelNavigator {
 from
   o : Before!GenNavigator
 to
   m : After!GenNavigator (
    contentExtensionID <- o.contentExtensionID,
    contentExtensionName <- o.contentExtensionName,
    contentExtensionPriority <- o.contentExtensionPriority,
    linkHelperExtensionID <- o.linkHelperExtensionID,
    sorterExtensionID <- o.sorterExtensionID,
    actionProviderID <- o.actionProviderID,
    contentProviderClassName <- o.contentProviderClassName,
    labelProviderClassName <- o.labelProviderClassName,
    linkHelperClassName <- o.linkHelperClassName,
    sorterClassName <- o.sorterClassName,
    actionProviderClassName <- o.actionProviderClassName,
    abstractNavigatorItemClassName <- o.abstractNavigatorItemClassName,
    navigatorGroupClassName <- o.navigatorGroupClassName,
    navigatorItemClassName <- o.navigatorItemClassName,
    uriInputTesterClassName <- o.uriInputTesterClassName,
    packageName <- o.packageName,
    childReferences <- o.childReferences
   )
}
rule GenNavigatorChildReference2GenNavigatorChildReference {
 from
   o : Before!GenNavigatorChildReference
 to
   m : After!GenNavigatorChildReference (
    parent <- o.parent,
    child <- o.child,
    referenceType <- o.referenceType,
    groupName <- o.groupName,
    groupIcon <- o.groupIcon,
    hideIfEmpty <- o.hideIfEmpty
```

```
    )
}
rule GenNavigatorPath2GenNavigatorPath {
  from
    o : Before!GenNavigatorPath
  to
    m : After!GenNavigatorPath (
      segments <- o.segments
    )
}
rule GenNavigatorPathSegment2GenNavigatorPathSegment {
  from
    o : Before!GenNavigatorPathSegment
  to
    m : After!GenNavigatorPathSegment (
      from <- o.from,
      to <- o.to
    )
}
rule GenPropertySheet2GenPropertySheet {
  from
    o : Before!GenPropertySheet
  to
    m : After!GenPropertySheet (
      tabs <- o.tabs,
      packageName <- o.packageName,
      readOnly <- o.readOnly,
      needsCaption <- o.needsCaption,
      labelProviderClassName <- o.labelProviderClassName
    )
}
abstract rule GenPropertyTab2GenPropertyTab {
  from
    o : Before!GenPropertyTab
  to
    m : After!GenPropertyTab (
      iD <- o.iD,
      label <- o.label
    )
}
rule GenStandardPropertyTab2GenStandardPropertyTab extends
    GenPropertyTab2GenPropertyTab {
  from
    o : Before!GenStandardPropertyTab
  to
    m : After!GenStandardPropertyTab
```

```
}
rule GenCustomPropertyTab2GenCustomPropertyTab extends
    GenPropertyTab2GenPropertyTab {
 from
  o : Before!GenCustomPropertyTab
 to
  m : After!GenCustomPropertyTab (
   className <- o.className,
   filter <- o.filter
  )
}
abstract rule GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!GenPropertyTabFilter
 to
  m : After!GenPropertyTabFilter
}
rule TypeTabFilter2TypeTabFilter extends
    GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!TypeTabFilter
 to
  m : After!TypeTabFilter (
   types <- o.types,
   generatedTypes <- o.generatedTypes
  )
}
rule CustomTabFilter2CustomTabFilter extends
    GenPropertyTabFilter2GenPropertyTabFilter {
 from
  o : Before!CustomTabFilter
 to
  m : After!CustomTabFilter (
   className <- o.className
  )
}
abstract rule GenContributionItem2GenContributionItem {
 from
  o : Before!GenContributionItem
 to
  m : After!GenContributionItem
}
rule GenSharedContributionItem2GenSharedContributionItem extends
    GenContributionItem2GenContributionItem {
 from
  o : Before!GenSharedContributionItem
 to
```

```
    m : After!GenSharedContributionItem (
      actualItem <- o.actualItem
    )
}
rule GenGroupMarker2GenGroupMarker extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenGroupMarker
  to
    m : After!GenGroupMarker (
      groupName <- o.groupName
    )
}
rule GenSeparator2GenSeparator extends GenContributionItem2GenContributionItem {
  from
    o : Before!GenSeparator
  to
    m : After!GenSeparator (
      groupName <- o.groupName
    )
}
rule GenActionFactoryContributionItem2GenActionFactoryContributionItem extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenActionFactoryContributionItem
  to
    m : After!GenActionFactoryContributionItem (
      name <- o.name
    )
}
abstract rule GenContributionManager2GenContributionManager extends
    GenContributionItem2GenContributionItem {
  from
    o : Before!GenContributionManager
  to
    m : After!GenContributionManager (
      iD <- o.iD,
      items <- o.items
    )
}
rule GenMenuManager2GenMenuManager extends
    GenContributionManager2GenContributionManager {
  from
    o : Before!GenMenuManager
  to
    m : After!GenMenuManager (
      name <- o.name
```

```
  )
}
rule GenToolBarManager2GenToolBarManager extends
    GenContributionManager2GenContributionManager {
 from
  o : Before!GenToolBarManager
 to
  m : After!GenToolBarManager
}
rule GenApplication2GenApplication {
 from
  o : Before!GenApplication
 to
  m : After!GenApplication (
   iD <- o.iD,
   title <- o.title,
   packageName <- o.packageName,
   className <- o.className,
   perspectiveId <- o.perspectiveId,
   supportFiles <- o.supportFiles,
   sharedContributionItems <- o.sharedContributionItems,
   mainMenu <- o.mainMenu,
   mainToolBar <- o.mainToolBar
  )
}
```

Listing C.13: GMF Generator migration in ATL

```
for (genLinkLabel in gen.GenLinkLabel.allInstances) {
 genLinkLabel.unset(notationViewFactoryClassName)
}

for (genLink in gen.GenLink.allInstances) {
 genLink.unset(notationViewFactoryClassName)
}

for (genEditorGenerator in gen.GenEditorGenerator.allInstances) {
 def genContextMenu = gen.GenContextMenu.newInstance()
 genEditorGenerator.contextMenus.add(genContextMenu)

 genContextMenu.context.add(genEditorGenerator.diagram)
 genContextMenu.items.add(gen.LoadResourceAction.newInstance())

 for (shortcutName in genContextMenu.diagram.containsShortcutsTo) {
  genContextMenu.items.add(gen.CreateShorcutAction.newInstance())
 }
```

```
}

for (genDiagram in gen.GenDiagram) {
 genDiagram.validationProviderPriority = gen.ProviderPriority#Lowest
}

for (featureLabelModelFacet in gen.FeatureLabelModelFacet) {
 def viewMethod = featureLabelModelFacet.unset(viewMethod)
 def editMethod = featureLabelModelFacet.unset(editMethod)
 featureLabelModelFacet.parser = createOrRetrievePredefinedParser(viewMethod,
    editMethod)
}

for (designLabelModelFacet in gen.DesignLabelModelFacet) {
 designLabelModelFacet.parser = createOrRetrieveExternalParser()
}


createOrRetrievePredefinedParser = { viewMethod, editMethod ->
 if (getPredefinedParser(viewMethod, editMethod) == null) {
   createOrRetrieveGenParsers().implementations.add(createPredefinedParser(
    viewMethod, editMethod))
 }

 return getPredefinedParser(viewMethod, editMethod)
}

getPredefinedParser = { viewMethod, editMethod ->
 return gen.PredefinedParser.allInstances.find{ it -> it.viewMethod ==
      viewMethod &amp;&amp; p.editMethod == editMethod }
}

createPredefinedParser = { viewMethod, editMethod ->
 def parser = gen.PredefinedParser.newInstance()
 parser.viewMethod = viewMethod
 parser.editMethod = editMethod
 return parser
}

createOrRetrieveExternalParser = {
 if (gen.ExternalParser.allInstances.size == 0) {
   createOrRetrieveGenParsers().implementations.add(gen.ExternalParser.
    newInstance())

 }

```

```
  return gen.ExternalParser.first
}

createOrRetrieveGenParsers = {
  if (gen.GenEditorGenerator.allInstances.first.labelParsers == null) {
    gen.GenEditorGenerator.allInstances.first.labelParsers = gen.GenParsers.newInstance()
    gen.GenEditorGenerator.allInstances.first.labelParsers.extensibleViaService = true
  }

  return gen.GenEditorGenerator.allInstances.first.labelParsers
}
```

Listing C.14: GMF Generator migration in Groovy-for-COPE

```
migrate GenLinkLabel {
  migrated.notationViewFactoryClassName := null;
}

migrate GenLink {
  migrated.notationViewFactoryClassName := null;
}

migrate GenEditorGenerator {
  migrated.contextMenus.add(new Migrated!GenContextMenu);
  migrated.contextMenus.first.context.add(migrated.diagram);

  migrated.contextMenus.first.items.add(new Migrated!LoadResourceAction);

  for (shortcutName in original.diagram.containsShortcutsTo) {
    migrated.contextMenus.first.items.add(new Migrated!CreateShortcutAction);
  }
}


migrate GenDiagram {
  migrated.validationProviderPriority := Migrated!ProviderPriority#Lowest;
}

migrate FeatureLabelModelFacet {
  migrated.parser := createOrRetrievePredefinedParser(migrated.viewMethod, migrated.editMethod);
  migrated.viewMethod := null;
  migrated.editMethod := null;
}

```

```
migrate DesignLabelModelFacet {
  migrated.parser := createOrRetrieveExternalParser();
}

operation createOrRetrievePredefinedParser(viewMethod : Any, editMethod : Any) :
      Migrated!PredefinedParser {
  if (getPredefinedParser(viewMethod, editMethod).isUndefined()) {
    createOrRetrieveGenParsers().implementations.add(createPredefinedParser(
     viewMethod, editMethod));
  }

  return getPredefinedParser(viewMethod, editMethod);
}

operation getPredefinedParser(viewMethod : Any, editMethod : Any) : Migrated!
     PredefinedParser {
  return Migrated!PredefinedParser.all.selectOne(p | p.viewMethod = viewMethod
      and p.editMethod = editMethod);
}

operation createPredefinedParser(viewMethod : Any, editMethod : Any) : Migrated!
     PredefinedParser {
  var parser := new Migrated!PredefinedParser;
  parser.viewMethod := viewMethod;
  parser.editMethod := editMethod;
  return parser;
}

operation createOrRetrieveExternalParser() : Migrated!ExternalParser {
  if (Migrated!ExternalParser.all.isEmpty()) {
    createOrRetrieveGenParsers().implementations.add(new Migrated!ExternalParser)
     ;
  }

  return Migrated!ExternalParser.all.first;
}

operation createOrRetrieveGenParsers() : Migrated!GenParsers {
  if (Migrated!GenEditorGenerator.all.first.labelParsers.isUndefined()) {
    Migrated!GenEditorGenerator.all.first.labelParsers := new Migrated!GenParsers
        ;
    Migrated!GenEditorGenerator.all.first.labelParsers.extensibleViaService :=
     true;
  }

  return Migrated!GenEditorGenerator.all.first.labelParsers;
```

```
}
```

Listing C.15: GMF Generator migration in Flock

# Appendix D

# TTC Results

This appendix describes the results of a model migration case submitted to the Tools Transformation Contest (TTC) 2010 workshop[1]. Nine solutions were submitted for the migration case and, during the workshop, the solutions were compared and awarded a score by the workshop participants and organisers. The results of the workshop are presented here, and were used for the evaluation of Epsilon Flock described in Section 6.4.

Below, each table presents scores for each of the nine solutions. The first seven tables show the score awarded to each tool for the criteria described in Section 6.4: correctness, conciseness, clarity, appropriateness, tool maturity, reproducibility and extensions. The final two tables show the total scores for each tool using an equal weight for each criterion, and using the weights determined by the workshop organisers. Each table shows the mean score (M column), variance between scores (V column), and the rank of each solution (R column). An `x` in a cell indicates a conflict of interest (no score awarded).

---

[1] `http://www.planet-research20.org/ttc2010/index.php?Itemid=132`

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PETE | 0 | 1 | 2 | 1 | x | 1 | 1 | 1 | x | 2 | 1 | 1 | 1.10 | 0.29 | 1 |
| COPE | 0 | 1 | 2 | 1 | x | 1 | 1 | 0 | 1 | 2 | 1 | 1 | 1.00 | 0.36 | 2 |
| Fujaba | 1 | x | 2 | 1 | 0 | 1 | 1 | 1 | 1 | x | 1 | x | 1.00 | 0.22 | 2 |
| GrGen | 0 | 1 | 2 | 1 | 1 | 1 | x | 1 | 1 | 2 | 0 | 1 | 1.00 | 0.36 | 2 |
| A/J | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0.83 | 0.14 | 5 |
| Mola | x | 1 | x | -1 | 0 | 1 | 1 | x | 1 | 2 | 1 | 1 | 0.78 | 0.62 | 6 |
| **Flock** | 0 | x | x | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0.70 | 0.21 | 7 |
| GReTL | 0 | x | x | 0 | 1 | x | 0 | 1 | 0 | 1 | 0 | 1 | 0.44 | 0.25 | 8 |
| RSDS | 0 | 1 | x | -1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0.18 | 0.33 | 9 |

Table D.1: Correctness scores (in the range -1 to 2).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 1 | x | x | 2 | 2 | 1 | 1 | 1 | 0 | 1 | 2 | 1 | 1.20 | 0.36 | 1 |
| COPE | 1 | 1 | 1 | 1 | x | 1 | 0 | 1 | 1 | -1 | 2 | 1 | 0.82 | 0.51 | 2 |
| GrGen | 0 | 2 | 1 | 1 | 1 | 0 | x | 1 | 0 | 1 | 1 | 1 | 0.82 | 0.33 | 2 |
| Fujaba | 1 | x | 1 | 0 | -1 | 0 | 0 | 0 | -1 | x | 1 | x | 0.11 | 0.54 | 4 |
| Mola | x | 0 | x | -2 | 0 | 0 | 0 | x | -1 | 1 | 1 | 1 | 0.00 | 0.89 | 5 |
| A/J | -1 | -1 | 1 | 0 | -1 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | -0.25 | 0.35 | 6 |
| PETE | -2 | -1 | 1 | 1 | x | -1 | -1 | 0 | x | 0 | 1 | -1 | -0.30 | 1.01 | 7 |
| GReTL | -1 | x | x | 1 | -1 | x | 0 | 0 | -1 | -1 | 0 | 0 | -0.33 | 0.56 | 8 |
| RSDS | 0 | -1 | x | -1 | -1 | 0 | 0 | -1 | -1 | 0 | -1 | 0 | -0.55 | 0.25 | 9 |

Table D.2: Conciseness scores (in the range -2 to 2).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 1 | x | x | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0.90 | 0.09 | 1 |
| COPE | 1 | 1 | 1 | 0 | x | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0.73 | 0.20 | 2 |
| GrGen | 0 | 0 | 0 | 1 | 0 | 1 | x | 0 | 1 | 1 | 0 | 1 | 0.45 | 0.25 | 3 |
| Mola | x | 1 | x | 1 | 0 | 1 | -1 | x | 0 | 1 | 0 | 1 | 0.44 | 0.47 | 4 |
| Fujaba | 1 | x | 0 | 1 | 0 | 0 | 0 | 0 | 0 | x | 1 | x | 0.33 | 0.22 | 5 |
| A/J | -1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0.25 | 0.35 | 6 |
| GReTL | 0 | x | x | 0 | 0 | x | 1 | 0 | 0 | 0 | 0 | 0 | 0.15 | 0.10 | 7 |
| PETE | 0 | 1 | 0 | 0 | x | 0 | 0 | 0 | x | 0 | 0 | 0 | 0.10 | 0.09 | 8 |
| RSDS | -1 | 1 | x | 0 | 0 | 0 | -1 | -1 | 0 | -1 | 1 | 0 | -0.18 | 0.51 | 9 |

Table D.3: Clarity scores (in the range -1 to 1).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 1 | x | x | 2 | 2 | 1 | 2 | 1 | 1 | 2 | 2 | 2 | 1.60 | 0.24 | 1 |
| COPE | 2 | 2 | 2 | 1 | x | 2 | 0 | 1 | 2 | 2 | 1 | 2 | 1.55 | 0.43 | 2 |
| GrGen | 0 | 1 | 1 | 1 | 0 | 1 | x | 1 | 1 | 1 | 0 | 1 | 0.73 | 0.20 | 3 |
| Mola | x | 1 | x | 0 | 0 | 1 | 0 | x | 0 | 1 | 1 | 2 | 0.67 | 0.44 | 4 |
| Fujaba | 1 | x | 1 | 0 | 0 | 1 | 0 | 0 | 0 | x | 0 | x | 0.33 | 0.22 | 5 |
| GReTL | 0 | x | x | 0 | 0 | x | 1 | 1 | 0 | 0 | 0 | 0 | 0.30 | 0.18 | 6 |
| PETE | -1 | 1 | 1 | 0 | x | 0 | 0 | 0 | x | 0 | 0 | 1 | 0.20 | 0.36 | 7 |
| A/J | 0 | 1 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 1 | 0 | 1 | 0.17 | 0.31 | 8 |
| RSDS | 0 | 0 | x | -1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0.09 | 0.26 | 9 |

Table D.4: Appropriateness scores (in the range -2 to 2).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fujaba | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | x | 0 | x | 0.89 | 0.10 | 1 |
| GrGen | 0 | 0 | 1 | 1 | 1 | 0 | x | 0 | 0 | 1 | 1 | 1 | 0.55 | 0.25 | 2 |
| **Flock** | 0 | x | x | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0.40 | 0.24 | 3 |
| Mola | x | 1 | x | 0 | -1 | 1 | 0 | x | 1 | 0 | 0 | 1 | 0.33 | 0.44 | 4 |
| COPE | 0 | 0 | 1 | 0 | x | 0 | 0 | -1 | 0 | 0 | 1 | 1 | 0.18 | 0.33 | 5 |
| PETE | -1 | 0 | 1 | 0 | x | 1 | 0 | -1 | x | 0 | 0 | 0 | 0.00 | 0.40 | 6 |
| GReTL | -1 | x | x | -1 | 0 | x | 0 | -1 | 0 | 0 | 0 | 0 | -0.28 | 0.23 | 7 |
| A/J | -1 | -1 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | -1 | 0 | -1 | -0.58 | 0.24 | 8 |
| RSDS | -1 | -1 | x | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 | 0 | -0.82 | 0.15 | 9 |

Table D.5: Tool maturity scores (in the range -1 to 1).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A/J | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| COPE | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| **Flock** | 1 | x | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| Fujaba | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | x | 1 | x | 1 | 0.00 | 1 |
| GReTL | 1 | x | x | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| GrGen | 1 | 1 | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| Mola | x | 1 | x | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 0.00 | 1 |
| PETE | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 0.00 | 1 |
| RSDS | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.00 | 9 |

Table D.6: Reproducibility scores (in the range 0 to 1).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 2 | x | x | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 1 | 1.80 | 0.16 | 1 |
| Fujaba | 1 | x | 2 | 2 | 2 | 2 | 2 | 1 | 1 | x | 1 | x | 1.56 | 0.25 | 2 |
| A/J | 1 | 2 | 2 | 2 | 0 | 2 | 2 | 2 | 2 | 1 | 0 | 2 | 1.50 | 0.58 | 3 |
| COPE | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1.00 | 0.00 | 4 |
| GReTL | 1 | x | x | 1 | 1 | x | 1 | 1 | 1 | 1 | 1 | 1 | 1.00 | 0.00 | 4 |
| GrGen | 1 | 1 | 1 | 1 | 2 | 1 | x | 1 | 1 | 1 | 0 | 1 | 1.00 | 0.18 | 4 |
| PETE | 1 | 1 | 1 | 1 | x | 1 | 1 | 1 | x | 1 | 0 | 0 | 0.80 | 0.16 | 7 |
| Mola | x | 0 | x | 0 | 0 | 0 | 0 | x | 0 | 0 | 1 | 0 | 0.11 | 0.10 | 8 |
| RSDS | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.00 | 0.00 | 9 |

Table D.7: Extensions scores (in the range 0 to 2).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 6 | x | x | 9 | 9 | 6 | 7 | 6 | 7 | 9 | 9 | 8 | 7.60 | 1.64 | 1 |
| COPE | 6 | 7 | 9 | 5 | x | 7 | 3 | 3 | 7 | 6 | 8 | 8 | 6.27 | 3.47 | 2 |
| GrGen | 2 | 6 | 7 | 7 | 6 | 5 | x | 5 | 5 | 8 | 4 | 7 | 5.64 | 2.60 | 3 |
| Fujaba | 7 | x | 8 | 6 | 3 | 6 | 5 | 4 | 3 | x | 5 | x | 5.22 | 2.62 | 4 |
| Mola | x | 5 | x | -1 | 0 | 5 | 2 | x | 2 | 6 | 5 | 7 | 3.44 | 6.91 | 5 |
| A/J | -1 | 3 | 5 | 3 | 1 | 5 | 4 | 3 | 4 | 3 | 2 | 4 | 3.00 | 2.67 | 6 |
| PETE | -2 | 4 | 7 | 4 | x | 3 | 2 | 2 | x | 4 | 3 | 2 | 2.90 | 4.69 | 7 |
| GReTL | 0 | x | x | 2 | 2 | x | 4 | x | 1 | 6 | 2 | 3 | 2.50 | 3.00 | 8 |
| RSDS | -2 | 0 | x | -4 | -2 | 0 | -2 | -3 | -1 | -2 | 1 | 1 | -1.27 | 2.38 | 9 |

Table D.8: Total (equally weighted) scores (in the range -7 to 11).

| **Tool** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **M** | **V** | **R** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Flock** | 17 | x | x | 28 | 28 | 20 | 20 | 17 | 23 | 29 | 28 | 26 | 23.60 | 20.64 | 1 |
| COPE | 17 | 22 | 31 | 16 | x | 22 | 10 | 7 | 22 | 21 | 26 | 26 | 20.00 | 45.45 | 2 |
| GrGen | 5 | 19 | 25 | 23 | 20 | 16 | x | 16 | 16 | 28 | 12 | 23 | 18.45 | 38.07 | 3 |
| Fujaba | 23 | x | 28 | 20 | 9 | 20 | 17 | 14 | 11 | x | 16 | x | 17.56 | 31.36 | 4 |
| Mola | x | 17 | x | -6 | -2 | 17 | 7 | x | 8 | 21 | 16 | 23 | 11.22 | 91.51 | 5 |
| PETE | -8 | 13 | 25 | 13 | x | 11 | 7 | 6 | x | 15 | 10 | 7 | 9.90 | 62.69 | 6 |
| A/J | -5 | 9 | 15 | 9 | 3 | 16 | 13 | 10 | 13 | 9 | 5 | 12 | 9.08 | 31.24 | 7 |
| GReTL | -2 | x | x | 4 | 7 | x | 11 | 9 | 2 | 12 | 5 | 10 | 6.44 | 18.91 | 8 |
| RSDS | -7 | 1 | x | -15 | -7 | -1 | -7 | -10 | -2 | -7 | 5 | 3 | -4.27 | 32.74 | 9 |

Table D.9: Total (weighted) scores (in the range -24 to 37).

# Bibliography

[Ackoff 1962] R.L. Ackoff. *Scientific Method: Optimizing Applied Research Decisions.* John Wiley and Sons, New York, 1962.

[Aizenbud-Reshef *et al.* 2005] N. Aizenbud-Reshef, R.F. Paige, J. Rubin, Y. Shaham-Gafni, and D.S. Kolovos. Operational semantics for traceability. In *Proc. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, pages 8–14, 2005.

[Alexander *et al.* 1977] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series).* Oxford University Press, New York, 1977.

[Álvarez *et al.* 2001] J. Álvarez, A. Evans, and P. Sammut. MML and the metamodel architecture. In *Proc. Workshop on Transformation in UML, co-located with the European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2001.

[Amyot *et al.* 2006] D. Amyot, H. Farah, and J.-F. Roy. Evaluation of development tools for domain-specific modeling languages. In R. Gotzhein and R. Reed, editors, *System Analysis and Modeling: Language Profiles*, volume 4320 of *Lecture Notes in Computer Science*, pages 183–197. Springer, Heidelberg, 2006.

[Apostel 1960] L. Apostel. Towards the formal study of models in the non-formal sciences. *Synthese*, 12(2):125–161, 1960.

[Arendt *et al.* 2009] T. Arendt, F. Mantz, L. Schneider, and G. Taentzer. Model refactoring in Eclipse by LTK, EWL, and EMF Refactor: A case study. In *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop, co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2009.

[Backus 1978] J. Backus. The history of FORTRAN I, II and III. *History of Programming Languages*, 1:165–180, 1978.

[Balazinska *et al.* 2000] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 98–107. IEEE Computer Society, 2000.

[Banerjee *et al.* 1987] J. Banerjee, W. Kim, H. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In U. Dayal and I.L. Traiger, editors, *Proc. Special Interest Group on Management of Data (SIGMOD) Conference*, pages 311–322. ACM Press, 1987.

[Barry 2006] B. Barry. A view of 20th and 21st century software engineering. In L.J. Osterweil, H.D. Rombach, and M.L. Soffa, editors, *Proc. International conference on Software Engineering (ICSE)*, pages 12–29. ACM, 2006.

[Beattie *et al.* 2007] B.R. Beattie, C.R. Taylor, and M.J. Watts. *The Economics of Production.* Krieger Publishing Company, 2nd edition, 2007.

[Beck & Cunningham 1989] K. Beck and W. Cunningham. Constructing abstractions for object-oriented applications. *Journal of Object Oriented Programming*, 2, 1989.

[Bézivin & Gerbé 2001] J. Bézivin and O. Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 273–280. IEEE Computer Society, 2001.

[Bézivin 2005] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[Biermann *et al.* 2006] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF model refactoring based on graph transformation concepts. *Electronic Communications of the European Association for the Study of Science and Technology [online]*, 3, 2006. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/34` [Accessed 2 November 2010].

[Bloch 2005] J. Bloch. How to design a good API and why it matters [online]. Keynote address to the LCSD Workshop at OOPSLA, October 2005, San Diego, United States of America. Available at: `http://lcsd05.cs.tamu.edu/slides/keynote.pdf` [Accessed 2 November 2010], 2005.

[Bohner 2002] S.A. Bohner. Software change impacts - an evolving perspective. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 263–272. IEEE Computer Society, 2002.

[Bosch 1998] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[Briand *et al.* 2003] L.C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 256–265. IEEE Computer Society, 2003.

[Brooks Jr. 1987] F.P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Brooks Jr. 1995] F.P. Brooks Jr. *The mythical man-month.* Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 25th anniversary edition, 1995.

[Brown *et al.* 1998] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *Anti Patterns.* Wiley, New York, 1998.

[Brun & Pierantonio 2008] C. Brun and A. Pierantonio. Model differences in the Eclipse Modelling Framework. *Upgrade*, 9(2):29–34, 2008.

[Cervelle *et al.* 2006] J. Cervelle, R. Forax, and G. Roussel. Tatoo: an innovative parser generator. In Ralf Gitzel, Markus Aleksy, and Martin Schader, editors, *Proc. International Symposium on Principles and Practice of Programming in Java (PPPJ)*, volume 178 of *ACM International Conference Proceeding Series*, pages 13–20. ACM, 2006.

[Chen & Chou 1999] J.Y.J. Chen and S.C. Chou. Consistency management in a process environment. *Systems and Software*, 47(2-3):105–110, 1999.

[Cicchetti *et al.* 2008] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proc. International IEEE Enterprise Distributed Object Computing Conference (EDOC)*, pages 222–231. IEEE Computer Society, 2008.

[Cicchetti 2008] A. Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering.* PhD thesis, Universita' degli Studi dell'Aquila, L'Aquila, Italy, 2008.

[Clark *et al.* 2008] T. Clark, P. Sammut, and J. Willians. *Superlanguages: Developing Languages and Applications with XMF [online].* Ceteva, Sheffield, 2008. Available at: `http://itcentre.tvu.ac.uk/~clark/Papers/Superlanguages.pdf` [Accessed 02 November 2010].

[Cleland-Huang *et al.* 2003] J. Cleland-Huang, C.K. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[Costa & Silva 2007] M. Costa and A.R. da Silva. RT-MDD framework – a practical approach. In *Proc. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, pages 17–26, 2007.

[Czarnecki & Helsen 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[de Lara & Guerra 2010] J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In D.C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and System (MoDELS), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2010.

[Deursen *et al.* 2000] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An annotated bibliography. *Special Interest Group on Programming Languages (SIGPLAN) Notices*, 35(6):26–36, 2000.

[Deursen *et al.* 2007] A. van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In *Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 41–49, 2007.

[Dig & Johnson 2006a] D. Dig and R. Johnson. Automated upgrading of component-based applications. In P.L. Tarr and W.R. Cook, editors, *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 675–676, 2006.

[Dig & Johnson 2006b] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.

[Dig *et al.* 2006] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[Dig *et al.* 2007] D. Dig, K. Manzoor, R. Johnson, and T.N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proc. International Conference on Software Engineering (ICSE)*, pages 427–436. IEEE Computer Society, 2007.

[Dig 2007] D. Dig. *Automated Upgrading of Component-Based Applications.* PhD thesis, University of Illinois at Urbana-Champaign, USA, 2007.

[Drivalos *et al.* 2008] N. Drivalos, R.F. Paige, K.J. Fernandes, and D.S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. Workshop on Traceability, co-located with the European Conference on Model Driven Architecture (ECMDA)*, 2008.

[Ducasse *et al.* 1999] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 109–118. IEEE Computer Society, 1999.

[Edelweiss & Moreira 2005] N. Edelweiss and Á.F. Moreira. Temporal and versioning model for schema evolution in object-oriented databases. *Data & Knowledge Engineering*, 53(2):99–128, 2005.

[Elaasar & Briand 2004] M. Elaasar and L.C. Briand. An overview of UML consistency management. Technical Report SCE-04-18, Carleton University, August 2004.

[Elmasri & Navathe 2006] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley Longman, Boston, Massachusetts, 5th edition, 2006.

[Erlikh 2000] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[Evans 2004] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley, Boston, Massachusetts, 2004.

[Feathers 2004] M.C. Feathers. *Working Effectively with Legacy Code.* Prentice Hall, Upper Saddle River, New Jersey, 2004.

[Ferrandina *et al.* 1995] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the O2 object database system. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 170–181. Morgan Kaufmann, 1995.

[Fowler 1999] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, Upper Saddle River, New Jersey, 1999.

[Fowler 2002] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, Massachusetts, 2002.

[Fowler 2010] M. Fowler. *Domain Specific Languages.* Addison-Wesley Professional, Boston, Massachusetts, 2010.

[Frankel 2002] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons Inc., New York, 2002.

[Fritzsche *et al.* 2008] M. Fritzsche, J. Johannes, S. Zschaler, A. Zherebtsov, and A. Terekhov. Application of tracing techniques in model-driven performance engineering. In *Proc. Traceability Workshop, co-located with the European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 111–120, 2008.

[Fuhrer *et al.* 2007] R.M. Fuhrer, A. Kiezun, and M. Keller. Refactoring in the Eclipse JDT: Past, present, and future. In *Proc. Workshop on Refactoring Tools (WRT), co-located with European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[Gamma *et al.* 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, Reading, Massachusetts, 1995.

[Garcés *et al.* 2009] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.

[Geiß & Kroll 2007] R. Geiß and M. Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE), Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569. Springer, 2007.

[Gosling *et al.* 2005] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java$^{TM}$ Language Specification.* Addison-Wesley, Boston, Massachusetts, 2005.

[Graham 1993] P. Graham. *On Lisp: Advanced Techniques for Common Lisp.* Prentice-Hall, Upper Saddle River, New Jersey, 1993.

[Greenfield *et al.* 2004] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons Inc., New York, 2004.

[Gronback 2009] R.C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, Boston, Massachusetts, 2009.

[Gruschko *et al.* 2007] B. Gruschko, D.S. Kolovos, and R.F. Paige. Towards synchronizing models with evolving metamodels. In *Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR)*, 2007.

[Guerrini *et al.* 2005] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. In A. Bonifati and D. Lee, editors, *Proc. International Workshop on Web Information and Data Management (WIDM)*, pages 39–44. ACM, 2005.

[Halstead 1977] M.H. Halstead. *Elements of Software Science.* Elsevier Science Inc., New York, 1977.

[Hearnden *et al.* 2006] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2006.

[Heidenreich *et al.* 2009] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2009.

[Henriksson *et al.* 2008] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3):165–184, 2008.

[Herrmannsdoerfer *et al.* 2008a] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Automatability of coupled evolution of metamodels and models in practice. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *Lecture Notes in Computer Science*, pages 645–659. Springer, 2008.

[Herrmannsdoerfer *et al.* 2008b] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE: A language for the coupled evolution of metamodels and models. In *Proc. International Workshop on Model Co-Evolution and Consistency Management (MCCM), co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2008.

[Herrmannsdoerfer *et al.* 2009a] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In S. Drossopoulou, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer, 2009.

[Herrmannsdoerfer *et al.* 2009b] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Proc. International Conference on Software Language Engineering (SLE), Revised Selected Papers*, volume 5696 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2009.

[Hussey & Paternostro 2006] K. Hussey and M. Paternostro. Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 02 November 2010] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`, 2006.

[ISO/IEC 1996] Information Technology ISO/IEC. Syntactic metalanguage – Extended BNF. ISO 14977:1996 International Standard, 1996.

[ISO/IEC 2002] Information Technology ISO/IEC. Z Formal Specification Notation – Syntax, Type System and Semantics. ISO 13568:2002 International Standard, 2002.

[Jackson 1995] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* ACM Press, New York, 1995.

[Jouault & Kurtev 2005] F. Jouault and I. Kurtev. Transforming models with ATL. In J-M. Bruel, editor, *Proc. Satellite Events at the International Conference on Model Driven Engineering Languages and Systems (MoDELS), International Workshops, Doctoral Symposium, Educators Symposium, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[Jouault 2005] F. Jouault. Loosely coupled traceability for ATL. In *Proc. Workshop on Traceability, co-located with the European Conference on Model-Driven Architecture (ECMDA)*, 2005.

[Jurack & Mantz 2010] S. Jurack and F. Mantz. Towards metamodel evolution of EMF models with Henshin. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Kalnins *et al.* 2005] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Proc. Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2005.

[Kataoka *et al.* 2001] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 736–743. IEEE Computer Society, 2001.

[Kelly & Tolvanen 2008] S. Kelly and J.P. Tolvanen. *Domain-Specific Modelling.* Wiley and IEEE Computer Society, 2008.

[Kelly 1999] T.P. Kelly. *Arguing Safety – A Systematic Approach to Safety Case Management.* PhD thesis, University of York, United Kingdom, 1999.

[Kerievsky 2004] J. Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[Kleppe *et al.* 2003] A.G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co. Inc., Boston, Massachusetts, 2003.

[Klint *et al.* 2003] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2003.

[Kolovos *et al.* 2006] D.S. Kolovos, R.F. Paige, and F.A. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[Kolovos *et al.* 2007] D.S. Kolovos, R.F. Paige, F.A.C. Polack, and L.M. Rose. Update transformations in the small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[Kolovos *et al.* 2008a] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. The grand challenge of scalability for model driven engineering. In M.R. Chaudron, editor, *Models in Software Engineering: Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 48–53. Springer-Verlag, 2008.

[Kolovos *et al.* 2008b] D.S. Kolovos, R.F. Paige, and F.A.C Polack. The Epsilon Transformation Language. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2008.

[Kolovos *et al.* 2009] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In J. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009.

[Kolovos *et al.* 2010] D.S. Kolovos, L.M. Rose, S. bin Abid, R.F. Paige, F.A.C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In D.C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and System (MoDELS), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2010.

[Kolovos 2009] D.S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management.* PhD thesis, University of York, United Kingdom, 2009.

[Kramer 2001] D. Kramer. XEM: XML Evolution Management. Master's thesis, Worcester Polytechnic Institute, MA, USA, 2001.

[Kurtev 2004] I. Kurtev. *Adaptability of Model Transformations.* PhD thesis, University of Twente, Netherlands, 2004.

[Lago *et al.* 2009] P. Lago, H. Muccini, and H. van Vliet. A scoped approach to traceability management. *Systems and Software*, 82(1):168–182, 2009.

[Lämmel & Verhoef 2001] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software - Practice and Experience*, 31(15):1395–1438, 2001.

[Lämmel 2001] R. Lämmel. Grammar adaptation. In J.N. Oliveira and P. Zave, editors, *Proc. Formal Methods for Increasing Software Productivity (FME), International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 2001.

[Lämmel 2002] R. Lämmel. Towards generic refactoring. In B. Fischer and E. Visser, editors, *Proc. ACM SIGPLAN Workshop on Rule-Based Programming*, pages 15–28. ACM, 2002.

[Lehman 1969] M.M. Lehman. The programming process. Technical report, IBM Research Report RC 2722, 1969.

[Lerner 2000] B.S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[Mäder *et al.* 2008] P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proc. IEEE International Requirements Engineering Conference (RE)*, pages 23–32. IEEE Computer Society, 2008.

[Martin & Martin 2006] R.C. Martin and M. Martin. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, Upper Saddle River, NJ, USA, 2006.

[McCarthy 1978] J. McCarthy. History of Lisp. *History of Programming Languages*, 1:217–223, 1978.

[McNeile 2003] A. McNeile. MDA: The vision with the hole? [online]. *Metamaxim Ltd*, 2003. [Accessed 02 November 2010] Available at: `http://www.metamaxim.com/download/documents/MDAv1.pdf`.

[Mellor & Balcer 2002] S.J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman, Boston, Massachusetts, 2002.

[Melnik 2004] S. Melnik. *Generic Model Management: Concepts and Algorithms.* PhD thesis, University of Leipzig, Germany, 2004.

[Méndez *et al.* 2010] D. Méndez, A. Etien, A. Muller, and R. Casallas. Towards transformation migration after metamodel evolution. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Mens & Demeyer 2007] T. Mens and S. Demeyer. *Software Evolution.* Springer-Verlag, Berlin, 2007.

[Mens & Tourwé 2004] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[Mens *et al.* 2007] T. Mens, G. Taentzer, and D. Müller. Challenges in model refactoring. In *Proc. Workshop on Object-Oriented Reengineering, co-located with the European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[Merriam-Webster 2010] Merriam-Webster. Definition of Nuclear Family [online]. [Accessed 02 November 2010] Available at: `http://www.merriam-webster.com/dictionary/nuclear%20family`, 2010.

[Moad 1990] J. Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–66, 1990.

[Moha *et al.* 2009] N. Moha, V. Mahé, O. Barais, and J.M. Jézéquel. Generic model refactorings. In A. Schürr and B. Selic, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5795 of *Lecture Notes in Computer Science*, pages 628–643. Springer, 2009.

[Muller & Hassenforder 2005] P. Muller and M. Hassenforder. HUTN as a Bridge between ModelWare and GrammarWare. In *Proc. Workshop in Software Modelling Engineering, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2005.

[Nentwich *et al.* 2003] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[Nguyen *et al.* 2005] T.N. Nguyen, C. Thao, and E.V. Munson. On product versioning for hypertexts. In *Proc. International Workshop on Software Configuration Management (SCM)*, pages 113–132. ACM, 2005.

[Nickel *et al.* 2000] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE)*, pages 742–745. ACM, 2000.

[Northrop 2006] L. Northrop. Ultra-large scale systems: The software challenge of the future. Technical report, Carnegie Mellon, June 2006.

[Oldevik *et al.* 2005] J. Oldevik, T. Neple, R. Grønmo, J.Ø. Aagedal, and A. Berre. Toward standardised model to text transformations. In A. Hartman and D. Kreische, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 3748 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2005.

[Olsen & Oldevik 2007] G.K. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In D.H. Akehurst, R. Vogel, and R.F. Paige, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 4530 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 2007.

[OMG 2001] OMG. Unified Modelling Language 1.4 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/spec/UML/1.4/`, 2001.

[OMG 2004] OMG. Human-Usable Textual Notation 1.0 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/technology/documents/formal/hutn.htm`, 2004.

[OMG 2005] OMG. MOF QVT Final Adopted Specication [online]. [Accessed 22 July 2009] Available at: `www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[OMG 2006] OMG. Object Constraint Language 2.0 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/technology/documents/formal/ocl.htm`, 2006.

[OMG 2007a] OMG. Unified Modelling Language 2.1.2 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/spec/UML/2.1.2/`, 2007.

[OMG 2007b] OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/spec/UML/2.2/`, 2007.

[OMG 2007c] OMG. XML Metadata Interchange 2.1.1 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/technology/documents/formal/xmi.htm`, 2007.

[OMG 2008a] OMG. Meta-Object Facility [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/mof`, 2008.

[OMG 2008b] OMG. Model Driven Architecture [online]. [Accessed 02 November 2010] Available at: `http://www.omg.org/mda/`, 2008.

[Opdyke 1992] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[Paige *et al.* 2007] R.F. Paige, P.J. Brooke, and J.S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3):1–48, 2007.

[Paige *et al.* 2009] R.F. Paige, L.M. Rose, X. Ge, D.S. Kolovos, and P.J. Brooke. FPTC: Automated safety analysis for domain-specific languages. In M.R.V. Chaudron, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2009.

[Parr 2007] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers, 2007.

[Patrascoiu & Rodgers 2004] O. Patrascoiu and P. Rodgers. Embedding OCL expressions in YATL. In *Proc. OCL and Model-Driven Engineering Workshop, co-located with the International Conference on the Unified Modeling Language (UML)*, 2004.

[Pilgrim *et al.* 2008] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. Constructing and visualizing transformation chains. In I. Schieferdecker and A. Hartman, editors, *Proc. European Conference on the Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2008.

[Pizka & Jürgens 2007] M. Pizka and E. Jürgens. Automating language evolution. In *Proc. Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 305–315. IEEE Computer Society, 2007.

[Porres 2003] I. Porres. Model refactorings as rule-based update transformations. In P. Stevens, J. Whittle, and G. Booch, editors, *Proc. International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML)*, volume 2863 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2003.

[RAE & BCS 2004] The RAE and The BCS. The challenges of complex IT projects. Technical report, The Royal Academy of Engineering, April 2004.

[Ramil & Lehman 2000] J.F. Ramil and M.M. Lehman. Cost estimation and evolvability monitoring for software evolution processes. In *Proc. Workshop on Empirical Studies of Software Maintenance (WESS), co-located with the International Conference on Software Maintenance*, 2000.

[Ráth *et al.* 2008] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live model transformations driven by incremental pattern matching. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 5063 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2008.

[Rios *et al.* 2006] E. Rios, T. Bozheva, A. Bediaga, and N. Guilloreau. MDD maturity model: A roadmap for introducing Model-Driven Development. In A. Rensink and J. Warmer, editors, *Proc. European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA)*, volume 4066 of *Lecture Notes in Computer Science*, pages 78–89. Springer Berlin / Heidelberg, 2006.

[Rising 2001] L. Rising, editor. *Design patterns in communications software.* Cambridge University Press, Cambridge, 2001.

[Rose *et al.* 2008a] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. Constructing models with the Human-Usable Textual Notation. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2008.

[Rose *et al.* 2008b] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. The Epsilon Generation Language. In I. Schieferdecker and A. Hartman, editors, *Proc. European Conference on Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[Rose *et al.* 2009a] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 545–549. IEEE Computer Society, 2009.

[Rose *et al.* 2009b] L.M. Rose, R.F. Paige, D.S. Kolovos, and F.A.C. Polack. An analysis of approaches to model migration. In *Proc. Joint Model-Driven Software Evolution and Model Co-evolution and Consistency Management (MoDSE-MCCM) Workshop, co-located with the International Conference on Model-Driven Engineering Languages and Systems (MoDELS)*, 2009.

[Rose *et al.* 2010a] L.M. Rose, A. Etien, D. Méndez, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Comparing model-metamodel and transformation-metamodel co-evolution. In *Proc. Models and Evolution Workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2010.

[Rose *et al.* 2010b] L.M. Rose, M. Herrmannsdoerfer, J.R. Williams, D.S. Kolovos, K. Garcés, R.F. Paige, and F.A.C. Polack. A comparison of model migration tools. In D.C. Petriu, N. Rouquette, and Ø Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.

[Rose *et al.* 2010c] L.M. Rose, D.S. Kolovos, N. Drivalos, J.R. Williams, R.F. Paige, F.A.C. Polack, and K.J. Fernandes. Concordance: An efficient framework for managing model integrity. In T. Kühne, B. Selic, M-P. Gervais, and F. Terrier, editors, *Proc. European Conference on Modelling Foundations and Applications*, volume 6138 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 2010.

[Rose *et al.* 2010d] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Migrating activity diagrams with Epsilon Flock. In *Proc. Transformation Tools Contest (TTC), co-located with the International Conference on Objects, Models, Components and Patterns (TOOLS Europe)*, 2010.

[Rose *et al.* 2010e] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model migration case. In *Proc. Transformation Tools Contest (TTC), co-located with the International Conference on Objects, Models, Components and Patterns (TOOLS Europe)*, 2010.

[Rose *et al.* 2010f] L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C Polack. Model migration with Epsilon Flock. In L. Tratt and M. Gogolla, editors, *Proc. International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

[Selic 2003] B. Selic. The pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[Selic 2005] B. Selic. Whats new in UML 2.0? *IBM Rational software*, 2005.

[Sendall & Kozaczynski 2003] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[Sjøberg 1993] D.I.K. Sjøberg. Quantifying schema evolution. *Information & Software Technology*, 35(1):35–44, 1993.

[Sommerville 2006] I. Sommerville. *Software Engineering.* Addison-Wesley, Boston, Massachusetts, 9th edition, 2006.

[Sprinkle & Karsai 2004] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, 2004.

[Sprinkle 2003] J. Sprinkle. *Metamodel Driven Model Migration.* PhD thesis, Vanderbilt University, TN, USA, 2003.

[Stahl *et al.* 2006] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, Chichester, 2006.

[Starfield *et al.* 1990] M. Starfield, K.A. Smith, and A.L. Bleloch. *How to model it: Problem Solving for the Computer Age.* McGraw-Hill Inc., New York, 1990.

[Steel & Raymond 2001] J. Steel and K. Raymond. Generating human-usable textual notations for information models. In *Proc. International Conference on Enterprise Distributed Object Computing (EDOC)*, pages 250–261. IEEE Computer Society, 2001.

[Steinberg *et al.* 2008] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, Boston, Massachusetts, 2008.

[Su *et al.* 2001] H. Su, D. Kramer, L. Chen, K.T. Claypool, and E.A. Rundensteiner. XEM: Managing the evolution of XML documents. In K. Aberer and L. Liu, editors, *Proc. International Workshop on Research Issues in Data Engineering (RIDE)*, pages 103–110. IEEE Computer Society, 2001.

[Tisi *et al.* 2009] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In R.F. Paige, A. Hartman, and A. Rensink, editors, *Proc. European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[Tratt 2008] L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–124, 2008.

[Varró & Balogh 2007] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[Vries & Roddick 2004] D. de Vries and J.F. Roddick. Facilitating database attribute domain evolution using meso-data. In S. Wang, D. Yang, K. Tanaka, F. Grandi, S. Zhou, E.E. Mangina, T.W. Ling, I-Y. Song, J. Guan, and H.C. Mayr, editors, *Proc. Conceptual Modeling for Advanced Application Domains, ER 2004 Workshops CoMoGIS, COMWIM, ECDM, CoMoA, DGOV, and ECOMO*, volume 3289 of *Lecture Notes in Computer Science*, pages 429–440. Springer, 2004.

[W3C 2007] W3C. W3C XML Schema 1.1 Specification [online]. [Accessed 02 November 2010] Available at: `http://www.w3.org/XML/Schema`, 2007.

[Wachsmuth 2007] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In E. Ernst, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, 2007.

[Wallace 2005] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

[Ward 1994] M.P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[Watson 2008] A. Watson. A brief history of MDA. *Upgrade*, 9(2):7–11, 2008.

[Welch & Barnes 2005] P.H. Welch and F.R.M. Barnes. Communicating mobile processes. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *Proc. Symposium on the Occasion of 25 Years Communicating Sequential Processes (CSP)*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer, 2005.

[Winkler & Pilgrim 2010] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9:529–565, 2010.